Empirical Software Engineering (2021) 26:74
https://doi.org/10.1007/s10664-021-09955-7

Lessons Learnt on Reproducibility in Machine Learning Based Android Malware Detection

Nadia Daoudi[1] · Kevin Allix[1] · Tegawendé F. Bissyandé[1] · Jacques Klein[1]

Accepted: 26 February 2021Published online: 24 May 2021

Abstract
A well-known curse of computer security research is that it often produces systems that, while technically sound, fail operationally. To overcome this curse, the community generally seeks to assess proposed systems under a variety of settings in order to make explicit every potential bias. In this respect, recently, research achievements on machine learning based malware detection are being considered for thorough evaluation by the community. Such an effort of comprehensive evaluation supposes first and foremost the possibility to perform an independent reproduction study in order to sharpen evaluations presented by approaches' authors. The question *Can published approaches actually be reproduced?* thus becomes paramount despite the little interest such mundane and practical aspects seem to attract in the malware detection field. In this paper, we attempt a complete reproduction of five Android Malware Detectors from the literature and discuss to what extent they are "reproducible". Notably, we provide insights on the implications around the guesswork that may be required to finalise a working implementation. Finally, we discuss how barriers to reproduction could be lifted, and how the malware detection field would benefit from stronger reproducibility standards—like many various fields already have.

Keywords Android malware dection · Machine learning · Reproducibility · Replicability

Communicated by: Meiyappan Nagappan

✉ Nadia Daoudi
nadia.daoudi@uni.lu

Kevin Allix
kevin.allix@uni.lu

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

Jacques Klein
jacques.klein@uni.lu

[1] Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, 29, Avenue J.F Kennedy, L-1855, Luxembourg, Luxembourg

I shall require that [the] logical form [of a scientific system] shall be such that it can be singled out, by means of empirical tests, in a negative sense: it must be possible for an empirical scientific system to be refuted by experience.
—Karl Popper, *The Logic of Scientific Discovery, 1959* (Popper 2002, p19)

1 Introduction

Mobile malware is now established as a serious threat. Its volume has increased steadily in recent years, in particular within the Android ecosystem whose popularity has attracted significant interest from malware writers. Security reports by Google, the Android platform maintainer, as well as by antivirus vendors even indicate that mobile malware is increasingly sophisticated and evolves fast, which suggests that organised criminal groups are heavily invested in producing and distributing malware at scale. To overcome this threat, the research community has quickly turned to machine learning techniques (Arp et al. 2014; Mariconti et al. 2017; Avdiienko et al. 2015; Garcia et al. 2018; Gascon et al. 2013; Jerome et al. 2014; Narayanan et al. 2017) as a potential panacea for identifying malware at scale within market apps and on users' devices. Approaches proposed in the literature have been reported to successfully identify a great deal of malware samples with little analysis effort (e.g., sometimes just enumerating requested permissions as learning features) and without any prior knowledge on the actual malicious behaviour to identify.

Recently, however, a number of studies have started to explicitly cast doubts on the performance achievements that are claimed in the literature. Notably, researchers have raised some concerns about the variety of evaluation biases that many works carry. For example, Allix et al. (2015) and Allix et al. (2016a) then (Pendlebury et al. 2019) have experimentally shown that the performance of malware detectors that are described in the literature is actually highly dependent on experimental parameters, such as dataset construction or evaluation methodology. Evaluation is indeed often biased towards reaching high performance while overlooking a general threat to validity in the fact that the machine learning paradigm itself brings its own share of variability in performance (Islam et al. 2017; Hutson 2018).

Evaluating the performance of a Security system is challenging (Van der Kouwe et al. 2019). Yet, to advance research on malware detectors, the community needs to ensure that evaluations of the approaches are comprehensive and reliable. Traditionally, research communities in various domains rely on independent reproduction to either adjust the level of trust in published results, and to uncover potential biases or limitations not considered in the original publication (Duvendack et al. 2015; Fokkens et al. 2013; Gundersen and Kjensmo 2018).

Reproducibility is an important criterion for acknowledging a research contribution. It is highly related to the concepts of *repeatability* and *replicability*. The exact meaning of these terms (i.e., *Repeatability*, *Reproducibility*, and *Replicability*) can however vary over time and across research fields (Plesser 2018). In the remainder of this paper we will refer to the *Terminology* section of ACM's *Artefact Review and Badging* policy (Association for Computer Machinery 2020)[1]. The concepts differ mainly on who re-performs the experiments and/or whether the experimental setup is changed from the original one:

- **Repeatability**: "Same team, same experimental setup";
- **Reproducibility**: "Different team, same experimental setup";
- **Replicability**: "Different team, different experimental setup".

[1] https://www.acm.org/publications/policies/artifact-review-and-badging-current, We note that this definition changed in 2020. In this paper we refer to the updated definition.

Given that the community must validate the assessment results provided by specific research groups in the literature, reproducibility appears to be the key concern.

Although there is a consensus on the benefits brought by reproduction, its conditions and practical barriers have so far received little attention in the field of Android malware detection. In recent years, the concerns have culminated to the extent that many major venues encourage the review of research artefacts in parallel to the submitted papers.

This paper We consider the possibility to reproduce machine learning based malware detection approaches by focusing on potentially influential research results presented at major venues. After applying filtering criteria (which are developed in Sect. 3.1), we have identified five approaches that are relevant for our empirical analysis on reproducibility of machine learning based Android malware detection approaches. Notably, our study investigates the following research questions:

- Can android malware detection approaches be reproduced and/or replicated?
- What is the amount of guesswork needed to reproduce these?
- What are the barriers to reproducibility?

To answer these questions, we embark on a complete reproduction attempt of the five identified literature approaches that rely on machine learning techniques to perform binary classification and family identification of Android apps:

- **DREBIN**, which was presented by Arp et al. (2014) at the NDSS symposium in 2014. With over a thousand citations, this work is regarded as a first breakthrough in this research direction.
- **MaMaDroid**, which was presented[2] by Mariconti et al. (2017) again at the NDSS symposium in 2017. This state of the art has also rapidly collected over a hundred citations.
- **RevealDroid**, which was published by Garcia et al. (2018) in the TOSEM journal in 2018. This work has been also presented at ICSE in the same year and has collected more than 50 citations.
- **DroidCat**, which was published in the TIFS journal in 2019 by Cai et al. (2019) and has collected more than 30 citations.
- **MalScan**, the most recent identified approach, which has been presented at ASE in 2019 by Wu et al. (2019).

Overall, our experiments have resulted in: (1) successfully reproducing MalScan approach; (2) successfully replicating DREBIN, MaMaDroid, and RevealDroid; (3) and failure to reproduce and replicate DroidCat

The paper reports on our following contributions:

- We discuss the implications of the practical obstacles that must be overcome when attempting to reproduce state-of-the-art approaches from the literature;
- We present a detailed explanation of why the reproduction attempts eventually turn into replication studies in machine learning based malware detection;
- We discuss how other scientific fields, when faced with similar questions, are trying to outgrow from what is often called the *Reproduction Crisis*;

[2]Note that an earlier version (Mariconti et al. 2016) of the research paper was published in 2016

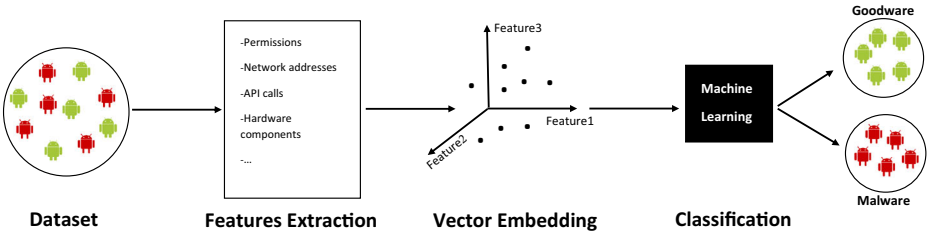

Fig. 1 Traditional machine learning building blocks

– We detail the consequences of non-reproducibility for the field of Android Malware Detection research.

2 Ingredients for Building a Machine Learning based Malware Detector

We revisit in this section the typical building blocks of an ML-based malware detector. The general workflow requires labelled datasets for training and testing ML models, a well-defined feature engineering procedure (features extraction + embedding) as well as a selection of supervised learning algorithms for classification. Figure 1 illustrates the various building blocks which are further detailed as follows.

2.1 Datasets

The first step in conducting a machine learning experiment is to collect a dataset. Such a dataset must be *relevant* to the prediction task (e.g., a dataset of apps with labels on maliciousness state is necessary to classify an android app as malware or goodware) and *complete* (e.g., the dataset must contain both malware and goodware). For the case of Android, labelled datasets can be collected from different sources:

– Curated malware datasets, such as *Genome* (Zhou and Jiang 2012) and *DREBIN* (Arp et al. 2014).
– Android app markets such as the official *Google Play Store*, and alternative markets like *Amazon Appstore* and *AppChina*.
– AndroZoo (Allix et al. 2016b), which is a growing collection of over 13 million Android apps regularly crawled from app markets.

For practicality reasons, in order to label app samples in a large scale manner, researchers rely on antivirus engines or online services such as `VirusTotal`[3] to determine if an app sample is malware or goodware. Depending on the level of consensus among different antivirus decisions, malware datasets may be more or less noised. For example, while many evaluation scenarios consider an app to be malicious as long as it is flagged by any antivirus, some researchers define a threshold of n (with $n > 1$) antivirus agreements for considering a sample as malware.

2.2 Feature Extraction

Once datasets are collected, researchers proceed to identify app features that must be extracted to build representations of each sample. Such feature engineering is focused

[3] https://www.virustotal.com hosts several antivirus products and other tools that regularly scan files to flag malicious samples

on making explicit the characteristics that can help discriminate malware from goodware samples. There is a variety of features that can be extracted from the APK files[4]. The most common ones in the literature are permissions, App components, intents, API calls, and network addresses. Various tools are then leveraged to perform feature extraction, including AAPT2 (Android Asset Packaging Tool)[5], AndroData (Khatter and Malik 2015), AppExtractor (Zhao et al. 2015), and AndroParse (Schmicker et al. 2019).

2.3 Embedding

Machine learning algorithms take as input numerical vectors that are amenable to computing in a vector space. Such vectors are constructed based on details of the extracted features. The idea of embedding is to create a vector space that translates the information contained in the features (e.g., a vector space that has the value 1 if the dimension (the feature) is present in the app, and 0 otherwise)

2.4 Classification

Once feature vectors are extracted for the samples in the dataset, learning algorithms can be applied to build classifiers. There are two types of classification:

- *Binary* classification, which is most prevalent, consists in predicting whether a sample belongs to the malware class or the goowdware class.
- *Multi-class* classification which promotes a finer-grained prediction, mainly to the level of which malware family the sample is associated to.

To evaluate classification approaches, a labelled dataset is required where each sample is known to be malware or goodware. Then, the dataset is split into:

- A *training* set, which is leveraged by the learning algorithms such as Support Vector Machines (Hearst et al. 1998) and Random Forests (Breiman 2001) to uncover the relationships between the input feature vectors, and the output class. Generally, the learning algorithm tries to discover the discrimination thresholds between classes in the vector space.
- A *test* set, which is separate from the training set, but drawn from the same distribution, is used to assess the performance of the trained classifier. Performance metrics are computed by comparing the predictions of the classifier on each test sample against the ground truth class of the sample.

Besides the datasets, which may impact the performance of the classifiers, some hyper-parameters tuning for the learning algorithms may significantly impact the resulting classifier.

Reproduction challenge. To perform reproduction experiments for a given literature approach, the aforementioned building blocks of the detector evaluation must be clearly described with sufficient details that leave no room to ambiguity. A lack of information in the description of any of the building blocks may deeply impact the reproducibility of the entire approach.

[4] APK (Android PacKage) if the file format used for Android applications
[5] https://developer.android.com/studio/command-line/aapt2

3 Reproduction

We now describe our journey to reproduce five approaches identified from the literature. First, we present the selection criteria, which eventually led to the identification of DREBIN (Arp et al. 2014), MaMaDroid (Mariconti et al. 2017), RevealDroid (Garcia et al. 2018), DroidCat (Cai et al. 2019), and MalScan (Wu et al. 2019). Then, for each reproduction subject, we (1) introduce the approach, (2) describe *how* we reproduce the building blocks of the machine learning approach, and (3) quantitatively compare the results obtained based on our reproduction against the results presented in the original publication. We note that we have contacted the original authors in order to ask them for some artefacts, and some problems encountered during the reproduction process.

Our reproduction is a contribution to the community effort for advancing the field of malware detection. It should not be viewed as a criticism of the reproduced approaches, nor as casting doubts on our community. This work has even been possible because original authors tried their best to help us with original dataset and even clarify code.

3.1 Reproduction Subject Selection

To select our candidate approaches, we have relied on the following methodology: First, we consider 16 major venues in Software Engineering, Security, and Machine Learning (EMSE, TIFS, TOSEM, TSE, FSE, ASE, ICSE, NDSS, S&P, Usenix Security, CCS, AsiaCCS, SIGKDD, NIPS, ICML, and IJCAI) that are consensually[6] tagged as the top venues in their respective domains.

Papers presented at these top conferences/journals in the last ten years (i.e., from 2009-2019) are then listed as the subject population. Then, we selected candidate papers that have {*malware* or *malicious*} and {*classification* or *detection* or *android* or *mobile*} in their title. Note that we have used these six keywords in order to maximise the chance to find the papers that deal with Android malware detection. Table 1 summarises the statistics of relevant subjects.

We then examined each of the 82 candidate papers individually and applied our first selection criteria as follows: (1) the paper must propose an approach (i.e., not about empirical analysis or surveys); (2) the paper must deal specifically with malware detection. 58 papers out of 82 have not satisfied these criteria, and then could not pass to the second round of selection.

Table 18 in Appendix includes justifications for the rejection of these 58 candidate papers based on the enumerated criteria.

The 24 remaining papers indeed matched our topic, and then passed to the second round of selection. At this stage, we examined again each paper individually, and we applied our second selection criteria as follows: (3) the paper must provide some artefacts to build on; (4) the paper must provide enough details about how to perform the experiments. Unfortunately, 19 papers out of 24 do not provide enough information about the dataset used in their experiments (or provide information only about a subset of the apps) and (or) do not share their artefacts (or provide enough details about their experiments) which makes it impossible to reproduce the approach. We present in Table 2 the details about the availability of the artefacts for these 19 papers.

[6]Influential papers - https://www.sec.cs.tu-bs.de/~konrieck/topnotch/; System security circus - http://s3.eurecom.fr/~balzarot/notes/top4_2019/; CORE ranking: http://portal.core.edu.au/conf-ranks/;

Table 1 Number of listed and candidate papers from the 16 venues

Name of the venue	Number of papers	
	listed	candidate
EMSE	645	2
TIFS	1906	20
ACM TOSEM	246	1
TSE	708	1
FSE	838	2
ASE	884	2
ICSE	1281	5
NDSS	583	10
IEEE S&P	533	2
USENIX Security Symposium	684	10
ACM SIGSAC CCS	1185	4
ACM Asia CCS	648	12
ACM SIGKDD	1890	7
NIPS	6089	1
ICML	3728	0
IJCAI	4958	3
TOTAL	26 806	82

In the end, six papers (out of 82) have satisfied the minimum criteria for the reproduction attempt. Our paper (Allix et al. 2016a) indeed matches our selection criteria, but we can not attempt its reproduction based on ACM terminology.

We present in Table 2 the details about the availability of the artefacts for the 24 papers that match our topic, including the 5 selected papers that are highlighted in yellow.

3.2 DREBIN

In 2014, Arp et al. (2014) have proposed an approach that performs a broad static analysis of Android apps to collect a large number of features for learning to discriminate malware from goodware. The analysis concerned both the disassembled bytecode as well as metadata from the Manifest file. Overall, their DREBIN approach uses 8 types of features: Hardware components, Requested permissions, App components, Filtered intents, Restricted API calls, Used permissions, Suspicious API calls, and Network addresses.

3.2.1 Dataset

To reproduce DREBIN's experiments, we obtained from the authors their original dataset of 5560 malware APK files, which represents 4.3% of the total number of apps used in their original publication. Unfortunately, the goodware dataset of 123 453 apks is not provided. Instead, the authors release the list of the apks' `SHA256` hashes. For our reproduction effort, we have leveraged AndroZoo (Allix et al. 2016b), which is to the best of our knowledge the largest collection of Android apps, to build up again the original goodware set. Unfortunately, after searching through the 10 millions of samples available at the time of the study in AndroZoo, we have managed to retrieve only a subset of 57 307 applications (i.e., only

Table 2 Short listed Papers. Yellow colour highlights the selected papers

Paper	Dataset						Features			Vector Emb			Classification			
	goodware hashes	goodware apps by authors	goodware apps found	malware hashes	malware apps by authors	malware apps found	code by authors	enough details	code imple-mented	code by authors	enough details	code imple-mented	code by authors	enough details	code imple-mented	Selection criteria*
Arp et al. (2014)	✓	×	p	✓	✓	✓	×	✓	✓	×	✓	✓	×	p	✓	✓
Mariconti et al. (2017)	✓	×	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	×	✓	×	✓
Zhang et al. 2014b	×	×	×	p	×	p	×	×	×	×	×	×	×	p	×	×
Zhu and Dumitraundefined (2016)	×	×	×	p	×	p	×	p	×	×	×	×	×	p	×	×
Zhou et al. (2012)	×	×	×	×	×	×	×	p	×	na	na	na	na	na	na	×
Wu et al. (2019)	✓	×	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	p	✓	✓
Chen et al. (2016)	×	×	×	p	×	p	×	p	×	×	p	×	×	p	×	×
Ye et al. (2019)	×	×	×	×	×	×	×	p	×	×	✓	×	×	p	×	×
Hou et al. (2018)	×	×	×	×	×	×	×	✓	×	×	p	×	×	p	×	×
Kim et al. (2019)	×	×	×	p	×	p	×	✓	×	×	p	×	×	p	×	×
Cai et al. (2019)	✓	×	p	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	p	✓	✓
Wang et al. (2018)	×	×	×	×	×	×	×	p	×	×	✓	×	×	p	×	×
Fan et al. (2018)	na	na	na	p	×	p	×	p	×	×	p	×	×	p	×	×

Table 2 (continued)

Paper	Dataset						Features			Vector Emb			Classification			
	goodware hashes	goodware apps by authors	goodware apps found	malware hashes	malware apps by authors	malware apps found	code by authors	enough details	code imple-mented	code by authors	enough details	code imple-mented	code by authors	enough details	code imple-mented	Selection criteria*
Sun et al. (2017)	×	×	×	p	×	p	×	p	×	na	na	na	na	na	na	×
Xu et al. (2016)	×	×	×	p	×	p	×	✓	×	×	p	×	×	p	×	×
Wang et al. (2014)	×	×	×	p	×	p	×	✓	×	×	p	×	×	×	×	×
Garcia et al. (2018)	✓	×	✓	✓	×	p	✓	p	✓	✓	✓	✓	✓	p	✓	✓
Canfora et al. (2019)	×	×	×	×	×	×	na	na	na	na	na	na	na	na	na	×
Hou et al. (2017)	×	×	×	×	×	×	×	p	×	×	p	×	×	✓	×	×
Narayanan et al. (2018)	×	×	×	p	×	p	×	p	×	×	p	×	×	✓	×	×
Allix et al. (2016a)	×	×	×	×	×	×	×	✓	✓	×	✓	✓	×	✓	✓	✓
Fan et al. (2019)	na	na	na	✓	×	✓	×	p	×	×	✓	×	×	p	×	×
Yang et al. (2018)	×	×	×	p	×	p	×	p	×	na	na	na	na	na	na	×
Yang et al. (2015)	×	×	×	p	×	p	×	p	×	×	✓	×	×	p	×	×

p = partially

* = Refer to our paragraph on selection criteria in Section 3.1

Table 3 DREBIN dataset

APK files	Malware	Goodware
Original Paper	5560	123 453
Provided by authors	5560	0
Obtained using SHA256 lists	0	57 307
Our completed dataset	0	66 153
After features extraction	5479	123 453

46.42% of the dataset used for the original experiments). Therefore, in order to keep the reproduction experiment as close as possible to the original experimental setup, we opted to complement the goodware dataset by selecting goodware samples in AndroZoo from the same period as indicated in the DREBIN publication (i.e., August 2010 - October 2012). Overall, Table 3 provides a summary of our collected dataset.

3.2.2 Feature Extraction

The feature set is presented by DREBIN's authors as a key contribution in their publication. They provide artefacts where different files already record the extracted features for their dataset apps. Given that we undertake to reproduce the entire approach, including the feature extraction process, we must re-run feature extractors on the collected dataset.

DREBIN uses four sets of features (hardware components, requested permissions, app components (names attributed by the developer to activities, services, content providers and broadcast receivers components), filtered intents) that are extracted from the manifest file, using the Android Asset Packaging Tool, and four other sets of features (restricted API calls, used permissions, suspicious API calls, network addresses) that are extracted from the disassembled code. Since the feature extraction scripts were not made available by the DREBIN authors, we relied on externally re-implemented extractors[7] (Narayanan et al. 2017). It is noteworthy that we opted to port the code from python2.7—which is now considered End of Life—to python3, and to use up-to-date libraries, in particular AndroGuard (Desnos and Gueguen 2011) that performs most of the APK processing. We have made sure that the porting has no impact on the output of the implemented code, and while not strictly required for the purpose of this reproduction, this software update will allow our own reproduction to outlive the already started phasing out of Python2.7.

After extracting the features, the size of the usable dataset is reduced, and passed from 5560 to 5479 apps for the malware, and from 57 307 to 57 300 apps for the goodware. This reduction is mainly due to the presence of invalid APK files. The goodware apps used to complete the goodware dataset are incrementally collected in order to have, after the features extraction, the total number of goodware samples that is used in the original publication. Our *final completed dataset* is then composed of 5479 malware applications, and 123 453 goodware samples as presented in Table 3

3.2.3 Feature Embedding

The extracted features are combined together to create vectors that are mapped into a n-dimensional vector space where n is the total number of the extracted features. Typically,

[7] https://github.com/MLDroid/drebin

given the n features extracted, if a feature exists in a sample apk, the vector associated to the apk will have the value 1 at the index corresponding to the feature. Otherwise, the value will be 0.

Although the original code to embed the extracted features into a vector-space amenable for machine learning is not available, the embedding process was adequately documented, hence the embedding component was readily re-implemented in previous work by Narayanan et al. (2017). Nevertheless, for practical reasons, we could only re-use the feature extraction scripts provided by Narayanan et al.. Indeed, Narayanan et al. re-used the feature set of DREBIN, but with a different experimental setup and the feature embedding step was too tightly coupled with their own setup to be re-used in the context of a full reproduction attempt.

3.2.4 Classification

DREBIN relies on Support Vector Machines (SVM) to predict the class (malware or goodware) of a given app. Like most machine learning algorithms, SVM has various options referred to as *hyper-parameters*. The relevance of these parameters is indirectly acknowledged in the original paper through the statement

> "The detection model and respective parameters of DREBIN are determined on the known partition [...]"(Arp et al. 2014)

No further mention is made about the actual hyper-parameter tuning that was performed. In particular, neither the values that were set nor the methods that were used to determine those values are documented. To work around this issue, Narayanan et al. had to come up with their own method of selecting C—the most important parameter of SVM—that may or may not be equivalent to what was done in the original work. We adopt our own strategy: after preliminary testing on more than 200 values of C, we noticed that C had in this case negligible impact for values of C comprised between 0.01 and 100. Thus, we opted to set $C = 1$, which is the default value implemented in scikit-learn[8], the popular Machine-Learning framework that we used for our reproduction experiments.

DREBIN authors indicate that 66% of their dataset has been used as a training set, and 33% as a test set. This procedure has been repeated 10 times and the results are averaged. The dataset partitioning and the way the performance metrics are computed is adequately documented, and we faced no difficulty when re-implementing this part.

The original publication was not accompanied with code artefacts on the classification part either, and did not provide a precise description of the implementation. We have implemented it ourselves, as part of the reproduction effort, using the scikit-learn framework.

3.2.5 Results

The results of our experiments show that DREBIN's approach reaches a recall score of 0.92. In the original publication, the model was reported to detect 94% of the malware. We provide in Fig. 2 the ROC curve extracted from DREBIN's publication, which represents the plot of the true-positive rate against the false positive rate. Our experiments lead to a slightly different ROC curve presented in Fig. 3. The reproduced model *slightly* outperforms the original one, but the overall performance is *similar*.

[8] https://scikit-learn.org

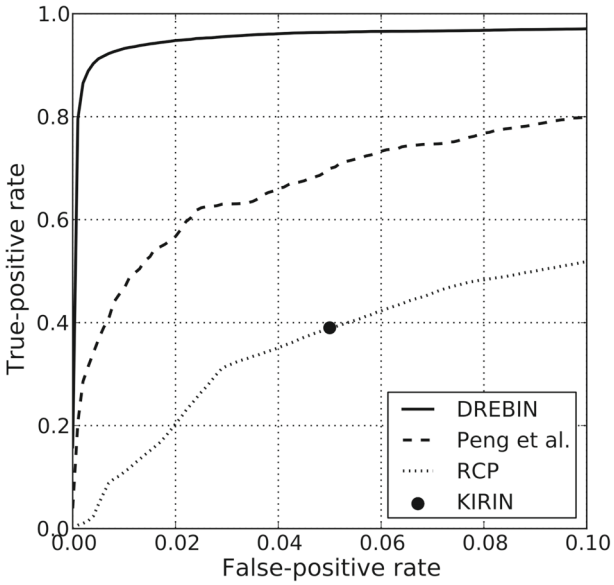

Fig. 2 ROC curve from DREBIN publication

3.3 MaMaDroid

In 2017, Mariconti et al. (2017) presented at the NDSS conference the MaMaDroid malware detection system, which attempts to characterise the application behaviour (Mariconti et al. 2017). An extended version (Onwuzurike et al. 2019) has even been recently published in the ACM Transactions on Privacy and Security. MaMadroid leverages the sequence of API calls that are performed by each app. In order to extract the sequence of API calls, a call graph is statically generated using the Soot (Vallée-Rai et al. 1999; Lam et al. 2011) static

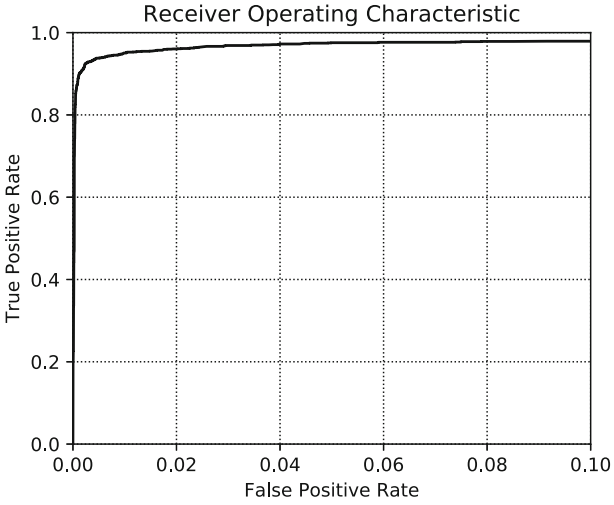

Fig. 3 ROC curve from reproduction experiments

analysis framework and FlowDroid (Arzt et al. 2014) for flow and context preservation. Concretely, extracted API calls are abstracted to either:

- *package*: the package name of the callee. E.g., `java.lang`
- or *family*: the first component of the package name of the callee. E.g., `java`, `android`, `google`

MaMaDroid then models the transitions between the abstracted API calls as Markov chains and uses a representation of those Markov chains as features for the machine learning prediction task.

3.3.1 Dataset

Experiments in the MaMaDroid publications are performed with a dataset consisting of 8447 goodware samples and 35 493 malware samples. The publication precisely describes the origin of the goodware set:

- 5879 apps are collected from PlayDrone (Viennot et al. 2014) in the period between April and November 2013. They form a subset referred to as `oldbenign`;
- 2568 apps are retrieved from the Google Play Store[9]. They are selected by considering "the top 100 apps in each of the 29 categories [...] as of March 7, 2016" (Mariconti et al. 2017). This subset is referred to as `newbenign`.

The malware dataset is split into five groups:

- 5560 malware apps coming from the DREBIN's paper (August 2010 - October 2012), denoted as `drebin`;
- `2013`: 6228 apps obtained from VirusShare[10];
- `2014`: 15 417 apps obtained from VirusShare;
- `2015`: 5314 apps obtained from VirusShare;
- `2016`: 2974 apps obtained from VirusShare.

Since the APK files are not directly provided by original authors, we have based our search on the lists of SHA256 hashes and the identified sources in order to retrieve the original apps. We provide in Table 16 in Appendix a detailed explanation of our dataset collection process.

Table 4 summarises the number of applications described in the original paper, the one provided in the lists of hashes, and the apps we have been able to download for `oldbenign`, `newbenign`, `drebin`, `2013`, `2014`, `2015`, and `2016` datasets.

3.3.2 Feature Extraction

MaMaDroid's authors make available the records of the extracted features from their experiments. Nevertheless, as for reproduction experiments of DREBIN, our goal is to achieve a full reproduction of MaMaDroid, including the feature extraction process. We thus discard the pre-computed features, and undertake to re-compute all features from the raw APKs.

Due to the presence of invalid APK files (Plain text, HTML, MS Word, ...), and analysis failures in Soot (as summarised in Table 5) during the process of extracting the features, the

[9] https://play.google.com/store

[10] https://virusshare.com is a repository of malware samples. It is maintained by one volunteer on his spare time

Table 4 MaMaDroid's dataset

Number of Applications	Paper	List of hashes	Available	After features extraction
oldbenign	5879	5879	5879	5410 (92.02%)
newbenign	2568	2555	1761	1446 (56.30%)
drebin	5560	5560	5560	5445 (97.93%)
2013	6228	11 080	11 080	6089 (97.76%)
2014	15 417	24 317	24 317	13 961 (90.55%)
2015	5314	5314	5216	4294 (80.80%)
2016	2974	2974	2974	2317 (77.90%)

static analyser could produce call graphs for only 5410, 1446, 5445, 6089, 13 961, 4294, and 2317 apps, for `oldbenign`, `newbenign`, `drebin`, `2013`, `2014`, `2015`, and `2016` respectively as described in Table 4.

Due to the low number of samples that could be considered after features extraction for `newbenign` (i.e., 56.30%), we decided to complete the dataset with apps from AndroZoo. Following the information in MaMaDroid, we select only apps until 7 March 2016.

The code to extract the features was made available by original authors. However, we noticed a potential problem with the way call graphs are processed by the provided code. This specific issue will be detailed in the discussion in Section 4, as we consider it to be a good illustration of the complexity that may arise from seemingly mundane details.

We had to adapt parts of the code so that it could run on a different directory structure than that of the original authors. We have also decided to port the code from python2 to python3, for the same reasons mentioned before. We believe that porting the code is not needed in order to perform our reproduction, but it will make it useful for other researchers that want to use DREBIN or MaMaDroid.

3.3.3 Feature Embedding

As described by its authors, MaMaDroid operates in two modes:

- In *package mode* (i.e., when API calls are abstracted to *package* names). A list of known packages provided by the Android Operating System is used to map API calls to their package name. Calls to code defined in an app (i.e., not part of the Android Framework) are abstracted to `self-defined`, while obfuscated code are mapped to `obfuscated`.
- In *family mode*, there are 11 possible families, i.e., `developer-defined`, `obfuscated`, `android`, `google`, `java`, `javax` , `xml`, `apache`, `junit`, `json`, `dom`. The last 9 families refer to `android.*`, `com.google.*`, `java.*`, `javax.*`, `org.xml.*`, `org.apache.*`, `junit.*`, `org.json`, and `org.w3c.dom.*` packages.

The code for the feature embedding is provided by authors. However, we noticed discrepancies between the provided code and the paper. While both the original paper (Mariconti

Table 5 Number of invalid APKs and number of failed feature extractions

Number of Applications	Invalid	Failed in feature extraction
MaMaDroid	200	17 825

et al. 2017) and its newer, extended version (Onwuzurike et al. 2019) explicitly state that the Android API version level considered is 24, the code instead uses level 28, or 26 in an older version of the code repository. The version considered has an impact on the list of packages, and potentially on families, known to the feature extractor program, and hence actually extracted. Since none of the two versions available to us matched the paper, we opted to use the latest version of the code that considers level 28.

Additionally, the extracted features as provided by original authors correspond to a filtering step that is described in Section 4.A of their paper (Mariconti et al. 2017), but the provided code does not perform this filtering. We therefore had to adapt this filtering step so that it matches the experiment described in the original paper.

Markov chains are built using the abstracted calls as *the states*, and the probabilities of moving from one state to another as *the transitions*. For each app, the feature vector is created using the probabilities of the transitions. States that are not present in a chain take the value of 0 in the feature vector. In family mode, each call is abstracted to one of the 11 families. Thus, there are 121 features representing the possible transitions in the markov chain. In package mode, the total number of the features stated in the paper is 115 600 and it represents the transitions between 340 states.

Principal Component Analysis (PCA) has also been used in several of MaMaDroid's experiments. It is a dimension-reduction tool, used to transform a large set of variables to a small set. The transformed set still contains most of the amount of information, and it is made of components that are linear combinations of the large set. Thus, PCA reduces the huge amount of memory needed to train the model and to perform the classification. The results of the experiments are reported using Precision, Recall, and f1 scores. In the paper, PCA is used with 10 selected components.

We note that the CSV files generated with authors' script do not match the expected output. In particular, when abstracting to family mode, all the feature vectors of the generated files are set to 0, and when abstracting to package mode, they are all set to 0 except the ones that represent transitions between obfuscated and selfdefined states. We have therefore examined the code of the authors, and we have made the necessary modifications so the calls are abstracted to the correct family/package. We note that we had also to modify the list of families provided in the code. In particular, we have changed `xml.`, `apache.`, `dom.`, and `json.` to `org.xml.`, `org.apache.`, `org.w3c.dom.`, and `org.json.` so the calls can be abstracted to the corresponding families, using the correct name of the package.

3.3.4 Classification

The Random Forests ensemble learning algorithm is leveraged in MaMaDroid for malware classification. In the original publication, the authors have explicitly discussed the values of the hyper-parameters. In *family* mode, MaMaDroid uses 51 trees with maximum depth of 8, while in *package* mode, they use 101 trees of maximum depth of 64. We note that MaMaDroid has actually been tested with 4 classification algorithms, namely Random Forests, 1-Nearest Neighbor (1-NN), 3-Nearest Neighbor (3-NN), and Support Vector Machines (SVM). However, since the best detection performances were achieved with Random Forests, the publication reports detailed performance metrics only for the Random Forest case. Therefore, our reproduction attempt is focused on the Random Forests-based classification.

The original NDSS publication also mentions that the 10-fold cross-validation technique has been used to provide the performance results of the prediction over different samples. Although the scripts for performing classification experiments were not made available by

Table 6 Performance of our reproduction attempt of MaMaDroid

Dataset \ Mode	[Precision, Recall, F-measure]			
	Family	Family (PCA)	Package	Package (PCA)
drebin, oldbenign	0.80 0.96 0.87	0.83 0.94 0.88	0.88 0.98 0.93	0.90 0.93 0.91
2013, oldbenign	0.87 0.94 0.90	0.91 0.92 0.91	0.96 0.96 0.96	0.94 0.93 0.93
2014, oldbenign	0.86 0.96 0.91	0.87 0.94 0.90	0.89 0.98 0.93	0.91 0.95 0.93
2014, newbenign	0.95 0.99 0.97	0.95 0.98 0.96	0.96 1.00 0.98	0.96 0.99 0.98
2015, newbenign	0.85 0.93 0.89	0.83 0.90 0.86	0.88 0.96 0.92	0.88 0.93 0.90
2016, newbenign	0.81 0.90 0.85	0.80 0.83 0.82	0.84 0.93 0.88	0.84 0.87 0.85

the authors, the publication contained enough information—including hyper-parameters—that allowed us to re-implement the necessary code.

3.3.5 Results

We have reproduced all 24 experiments from MaMaDroid where authors provided detailed performance metrics. These experiments correspond to the section IV.B of reference (Mariconti et al. 2017), and evaluate the approach when the training set and the test set come from the same period. We did not attempt to reproduce the other experiments of MaMaDroid since they are either focusing on a specific point (sensitivity to the number of years separating the training set from the test set) or comparing to another approach. The 24 reproduced experiments correspond to the evaluation of the approach over 6 different combinations of training and test sets and 4 different *modes* (namely, Family mode without PCA; Family mode with PCA, Package mode without PCA, and Package mode with PCA).

Performance results are presented in terms of Precision, Recall, and F-measure scores. We opt to present the results of the 24 experiments in a table that has the same design of the original paper. We thus provide the metrics' scores for our reproduced experiments in Table 6, and the scores provided by the original authors in Table 7. The first column of the tables represents the pairs of datasets (malware and goodware) trained and tested using 10-fold cross-validation technique. From the two Tables, we can notice that the Precision of original paper for all the experiments (with and without PCA) is higher in 23 out

Table 7 Performance of MaMaDroid, as reported by original authors: (Onwuzurike et al. 2019)

Dataset \ Mode	[Precision, Recall, F-measure]			
	Family	Family(PCA)	Package	Package (PCA)
drebin, oldbenign	0.82 0.95 0.88	0.84 0.92 0.88	0.95 0.97 0.96	0.94 0.95 0.94
2013, oldbenign	0.91 0.93 0.92	0.93 0.90 0.92	0.98 0.95 0.97	0.97 0.95 0.96
2014, oldbenign	0.88 0.96 0.92	0.87 0.94 0.90	0.93 0.97 0.95	0.92 0.96 0.94
2014, newbenign	0.97 0.99 0.98	0.96 0.99 0.97	0.98 1.00 0.99	0.97 1.00 0.99
2015, newbenign	0.89 0.93 0.91	0.87 0.93 0.90	0.93 0.98 0.95	0.91 0.97 0.94
2016, newbenign	0.87 0.91 0.89	0.86 0.88 0.87	0.92 0.92 0.92	0.88 0.89 0.89

Table 8 RevealDroid's dataset

APK files	Goodware	Drebin	VirusShare	VirusTotal
Original Paper	24 679	5538	22 592	2152
Downloaded	24 999	5559	22 436	269
After the features extraction	23 897	5544	22 211	189

of 24 experiments (with a largest difference of 0.08, i.e., up to 8 percentage points), and equal to our results in 1 experiment. The Recall of original paper is higher in 11 out of 24 experiments (with a largest difference of 0.05, i.e., up to 5 percentage points), smaller in 8 experiments (with a largest difference of 0.02), and equal to our Recall in 5 experiments. As for F-measure of the paper, it is higher in 22 out of 24 experiments (with a largest difference of 0.05, i.e., up to 5 percentage points), and equal to our results in 2 experiments.

3.4 RevealDroid

RevealDroid is a machine-learning-based Android malware detection and family identification approach that was published in 2018 in the TOSEM journal. The authors also presented their approach at the ICSE conference in the same year as a journal first paper. Overall, RevealDroid extracts features that belong to three categories: Android-API Usage, Reflective Feature, and Native Calls.

3.4.1 Dataset

RevealDroid is evaluated using benign apps collected from AndroZoo, and malicious apps collected from the following 4 sets: Genome project, DREBIN's dataset, VirusShare, and VirusTotal. To collect their dataset, we have downloaded the compressed file provided by the authors in the RevealDroid dedicated webpage[11]. However, the provided file does not contain the raw APK files needed for our reproduction. Instead, it contains 1065 files that are mainly related to the extracted features.

Fortunately, we discover in the RevealDroid code repositories [12][13] files containing lists of hashes of the apps used in the original RevealDroid paper. We present in Table 8 the number of apps used in the original paper, as well as the number of apps we were able to collect for the four sets (we do not present the Genome dataset in the table since it is already included in Drebin dataset). A detailed explanation of the dataset collection process is provided in Table 9.

3.4.2 Feature Extraction

RevealDroid uses three sets of features:

– Android API-Usage: the features considered are: (1) the number of Android API method invocations, noted MAPI, (2) the number of API invocations for specific

[11] https://seal.ics.uci.edu/projects/revealdroid/
[12] https://bitbucket.org/joshuaga/revealdroid/src/master/
[13] https://bitbucket.org/joshuaga/android-reflection-analysis/src/master

Table 9 RevealDroid's dataset collection process

Dataset	Description
Benign	The authors have used 24 679 benign apks, but when searching in the code repositories of RevealDroid [a], we have found two potential benign lists of hashes that contain 24 996 and 24 999 entries respectively. Since none of these lists matched the number of apps cited in RevealDroid's paper, we have decided to use the list that contain 24 999 hashes, in order to ensure that we have enough benign apps if some of them failed in the features extraction process. We have then leveraged AndroZoo to retrieved these apps as pointed out by the authors in the paper, and we have successfully downloaded all the 24 999 APKs.
VirusShare	For VirusShare dataset, we have found again two lists of VirusShare hashes with 23 131 and 22 592 apps respectively. Luckily the number of apps in the second list matched exactly the number of apps used in the paper (22 592). To collect these apps, we have contacted the authors to advise us on the exact VirusShare torrents that contain the apps used in their experiments. However, we did find only a subset of the apps in the named torrents (we suppose that VirusShare has changed the content of their torrents after the original authors have retrieved the apps). Thus we had to search over all the torrents and we have successfully downloaded the needed apps that are distributed over 65 torrents. We note that during the process of our reproduction, we have found that 156 apps from VirusShare (identified with md5) are duplicated in Drebin's dataset (identified with sha256). We have thus removed these apps from VirusShare set.
VirusTotal	For VirusTotal apps, unfortunately we were unable to identify the exact list of hashes even with the advise of the original authors. As a workaround, we have found a list of hashes with their family labels that helped us identify the missing hashes after excluding DREBIN's and VirusShare's. The list contains 30 868 hashes, 554 of them are duplicated entries. After excluding the known apps and the duplicated ones, we ended up with 1723 md5 and 429 sha256 that are potentially the apps used in VirusTotal set. Again, during our experiments, we have found that 2 apps identified with md5 from VirusTotal are actually DREBIN's apps, and 8 apps identified with sha256 from VirusTotal are actually VirusShare's apps. Using AndroZoo, we have been able to collect only 269 (out of 2151) APK files.
drebin	We note that for this dataset, the authors use 5538 apps of the original collection (5560). In order to identify the needed apps, we have found two potential files that contain 5560 and 5654 hashes respectively. Since none of them matched the exact number used, we have decided to use the apps that are included in the list of families that helped us identify VirusTotal apps (5559 apps).
Genome	It is included in DREBIN's dataset

[a]https://bitbucket.org/joshuaga/revealdroid/src/master/ and https://bitbucket.org/joshuaga/android-reflection-analysis/src/master

Android API packages, noted PAPI. These features are extracted using Dexpler (Bartel et al. 2012) and Soot (Vallée-Rai et al. 1999);

- Reflective Feature: RevealDroid extracts information about dynamic class loading that is represented by: the full or partial method names invoked; the number of times the full or partial method name is invoked; and the total number of reflective invocations. Note that full method names are defined by the original authors as the methods that have both the reflectively invoked method and class names statically determined, and the partial method names are the ones that have only the invoked method name statically determined. These features are also extracted using Dexpler and Soot.
- Native Call: These features are represented by the external calls of every binary in the app, and the number of invocations of each external call. Native Call features are extracted based on the Android ABI toolchain[14].

[14]https://developer.android.com/ndk/guides/abis

The source code of the features extractor is made available by the authors of RevealDroid. However, we have faced some issues compiling their tool due to the mismatch between the version of some libraries present in the repository with the version required by the code. Also, we were not sure about the exact script that extracts Android API-Usage features. The documentation of the repository shows two potential scripts that are used to extract sensitive API features and package API features respectively. Confused about sensitive API features extractor (if it needs to be used) and if the script that extracts MAPI features is missing, we have decided to contact original authors. With their help, we have made sure that the package API features extractor is all what we need to use. We have also validated with original authors the relevant output of this extractor since it generates two outputs.

We note that we were not able to extract the features for all the apps in our collection, because Soot has crashed on some apps. We present in Table 8 the number of apps we have ended up with after extracting the features.

3.4.3 Feature Embedding

The three sets of features presented in the previous section with their values are used to construct a feature vector for each app. The code for the feature embedding was to some extent difficult to identify among the variety of scripts the authors provide. However, searching over the entire two repositories helped us to find some useful scripts that served as a base for our vector embedding script. Specifically, we have noticed that the classification's code requires as input either a CSV or an HDF file that stores the features matrix. Thus we have written the script in a way to generate an HDF file output for each dataset so they can be concatenated later (using a script provided by original authors) and thus used as input in the classification task. However, after running our script for the benign dataset, we have encountered some problems related to the storage space needed to generate the HDF file (more than 200GB for the benign dataset only). Hence, we have re-written another script that does not store the matrix but uses it directly to perform the classification.

3.4.4 Classification

As we have mentioned previously, RevealDroid is a malware detector but also a family identifier. In the following, we present each of the working modes separately.

Malware detection In this mode, RevealDroid is evaluated for its ability to distinguish malware and benign apps using linear SVM and 10-fold cross-validation setting. We note that the hyper-parameters of the classifier are not mentioned in the paper. However, thanks to the code made available by the authors, we were able to know that RevealDroid does not use the default hyper-parameters for C, $penalty$, and $dual$, that are set to 0.01, $l1$, and $False$ respectively. We have thus combined the authors' code (and ported it from python2 to python3) with our script of the previous step in order to avoid the storage issues. We have also noticed that the matrix used in the cross-validation does not involve any shuffling. Since the authors' script takes as input an HDF file (that we do not know how exactly it has been generated), we are not sure if the matrix stored in that file is already shuffled or not. Thus we have added this step to the script since we know that our matrix is not shuffled.

Family detection RevealDroid is able to identify the family label of malware apps using a Classification and Regression Trees (CART) classifier. The evaluation is made using 10-fold cross-validation and the classifier is used with its default hyper-parameters. The number of

Table 10 Performance of RevealDroid in malware detection

	Original results			Our results		
	Precision	Recall	F1	Precision	Recall	F1
Benign	98%	97%	98%	95%	86%	90%
Malicious	98%	98%	98%	89%	96%	92%
Average	98%	98%	98%	92%	91%	91%

labels in this mode of operation is determined by the number of malware families contained in the training set. Similarly to malware detection, we had to write our own script to perform the classification.

Searching over the code repository, we have identified a file that contains 30 868 hashes with their respective family labels. Thus, we have used this file in order to map each malware in our collected dataset to its family label.

3.4.5 Results

We have reproduced the experiments that assess how accurate RevealDroid is for both the malware detection task and the family identification task. We did not attempt to reproduce the other experiments for the same reasons mentioned for MaMaDroid's reproduction.

Malware detection The entire dataset has been used to evaluate RevealDroid in a time agnostic scenario where the age of the apps is not taken into consideration when determining the training and testing sets. Original authors have reported their results with Precision, Recall, and F1 score. We present in Table 10 our results compared to original authors results.

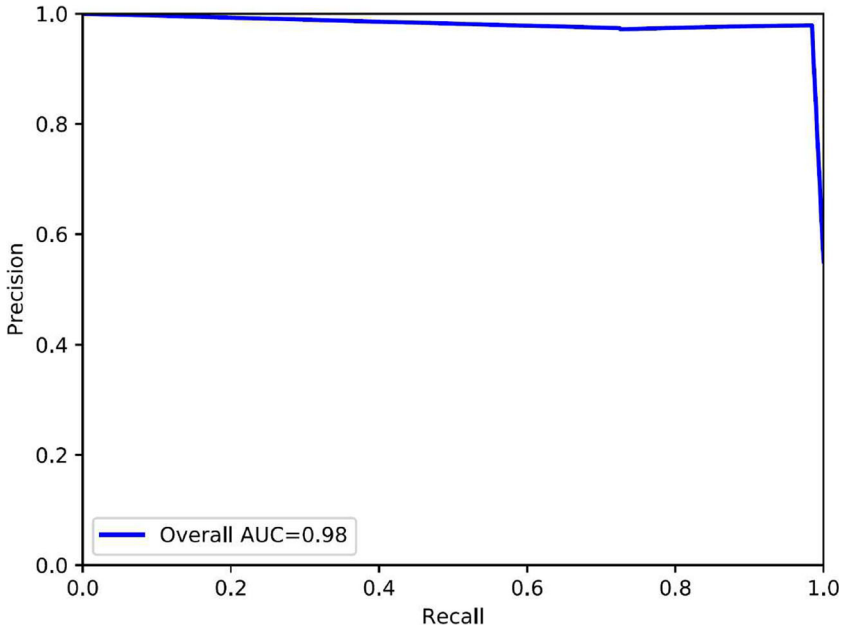

Fig. 4 Precision Recall Curve from RevealDroid's publication

We notice that the metrics reported in RevealDroid's paper are all higher than the ones we have obtained, with a largest difference of 11 percentage points for benign identification, and a largest difference of 9 percentage points for malware detection.

RevealDroid's authors also provide the precision-recall (PR) curve of their approach. This curve represents the precision and recall values for different probability thresholds. We provide in Fig. 4 the precision-recall curve extracted from RevealDroid's publication. Our experiments show again a different PR curve presented in Fig. 5. Our PR curve that has an area under the curve (AUC) of .96 is not as good as the one reported by original authors which has an AUC of .98

Family detection The original publication evaluates RevealDroid performance for family detection using two experiments that involve two datasets:

- The first evaluation is performed on the Genome dataset. The publication reports only the number of family labels found in this dataset (48 malware families) but not the exact number of apps selected to perform this experiment. Our genome dataset (collected based on the list of hashes from RevealDroid's repository) contains 1232 apps with 41 malware family labels. The authors report that their model has 95% correct classification rate. Our reproduction results show a correct classification rate of 91%.
- The second evaluation involves malware apps from DREBIN, VirusShare, and VirusTotal datasets. In this experiment, original authors have used 27 979 malware samples with 447 family labels. Our collected dataset contains 26 084 malware apps with 388 families. Original results show a correct classification rate of 84%. Our reproduction reports a correct classification rate of 85%.

3.5 DroidCat

DroidCat is a malware detection and categorisation approach that has been published on 2019 in the TIFS journal. DroidCat is based on dynamic analysis along with machine learning techniques. The tool uses 70 features that are related to the structure of apps executions, Inter-Component Communication (ICC), and security sensitive accesses. These features

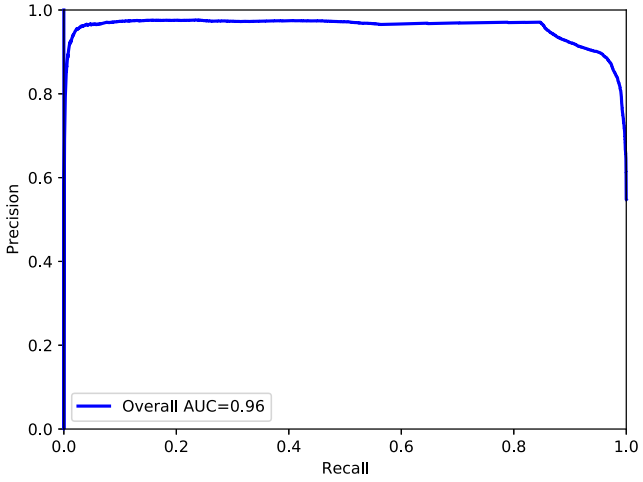

Fig. 5 Precision Recall Curve from reproduction experiments

Table 11 DroidCat dataset

		Benign				Malware			
Dataset	Period	source	original	lists	collected	source	original	lists	collected
D1617	2016-2017	GP,AZ	5346	2982	2724	VS,AZ	3450	3653	3653
D1415	2014-2015	GP,AZ	6545	4131	4131	VS,AZ	3190	2788	2788
D1213	2012-2013	GP,AZ	5035	3413	3413	VS,AZ,DB,MG	9084	3084	3084
D0911	2009-2011	AZ	439	3101	3101	VS,AZ,DB,MG	1254	4308	4308

have been selected based on a benchmark suit that involved 122 features belonging to the same mentioned categories.

3.5.1 Dataset

DroidCat's experiments involve android apps from different sources covering the period from 2009 to 2017. The malware dataset is collected from AndroZoo (AZ), VirusShare (VS), Drebin (DB), and Genome dataset (MG). The benign apps are downloaded from AndroZoo and Google Play (GP). The authors provide the lists of hashes on their website[15], and state that the apps can be downloaded from AndroZoo. In total, we need to collect eight datasets: four collections for malware and four for goodware. While examining the lists of the hashes, we have noticed that the numbers of apps do not match the ones reported in DroidCat publication. Specifically, we had missing apps in five out of eight datasets, and more apps in the three remaining sets. We have thus contacted the authors to advise us on how to obtain the missing hashes and what are the apps that should be discarded in order to have the same exact experimental setup. Unfortunately, we did not get a response to our email. Thus, we decided to build the dataset based on the available lists of hashes, even if several apps from the original paper are missing. We have successfully downloaded almost all the apps from AndroZoo, except for 2017 benign dataset that we have retrieved from Google Play. We present in Table 11 a summary of the original and our collected dataset.

3.5.2 Feature Extraction and Embedding

DroidCat is based on dynamic analysis and machine learning techniques to detect android malware apps and their families. To extract the features, DroidCat instruments an app using Soot (Vallée-Rai et al. 1999) in order to collect its execution traces when run in an Android emulator. The execution traces include method calls and ICCs. DroidCat's authors leverage Monkey's[16] randomly generated inputs in order to automatically collect these traces.

After collecting the traces, DroidCat extracts 70 features that belong to three categories:

- Structure: It contains 32 features that describe the distribution (i.e., percentage and frequency) of method calls, their declaring classes, and caller-callee links.
- ICC: It contains 5 features to describe the ICC distributions. For instance: the percentage of external explicit ICCs.

[15] https://chapering.github.io/droidcat/

[16] https://developer.android.com/studio/test/monkey

– Security: It includes 33 features that describe the distribution of sources and sinks, as well as the reachability between them.

These features computation is implemented on top of the authors' Android App Dynamic Characterisation Toolkit DroidFax[17], and their Android data-flow analysis and instrumentation library duafdroid[18], which is based on the Soot framework. The code for features extraction is made available by DroidCat's authors, but still we have faced some issues with Soot that crashes either during the apps instrumentation or the features computation. We had thus to solve the issues by using a more recent android jar file (API level 30) than the one used by the authors (API level 19). We also had to handle the exceptions that occurred for some methods, which prevented DroidCat to compute the features. DroidCat features extractor generates, for each app, a vector that contains the package name of the app, and the values of the computed features.

3.5.3 Classification

DroidCat trains a Random Forest classifier with 128 trees, for both malware detection and family identification. DroidCat is evaluated on each of the four datasets separately. For each dataset, the apps of each class are sorted by their age (first-seen date obtained from VirusTotal) and are split at 70 percentile. The performance of DroidCat is thus evaluated on the 30% newest apps, the remaining 70% apps being used for the training.

The code for the classification part is also made available by the authors. However, we have found a variety of scripts that perform the classification differently (random split, cross-validation, ...). We have thus selected randomly one of the scripts that split the dataset by their first-seen date (we note these scripts date-scripts), and we have ported it from Python2 to Python3. But again, we have noticed that the date-scripts extract the date from the features vectors themselves. We were puzzled by this information since the features vectors generated in the last step contain only the name of the package and the features' values. While searching again in the repository, we have found a script that adds the date to the features vectors of each app, but this script assumes that the feature vectors are identified with their sha256 hashes and not with their package name (so it can find the first-seen date in other files found in the same repository). Since the package name does not enable to uniquely identify the apps, we have decided to re-compute the features and modify the code of the authors so it generates the correct output.

For family classification we have used the labels files that map directly malware hashes to their families, which is available in DroidCat's repository as well as the script that loads family labels from these files. The other scripts select the family label from VirusTotal reports, which we could not find in the repository. However, we have noticed some issues with the selected script: (1) Apps for which it could not find a label are attributed the MALICIOUS label. (2) Some samples have "none" as their family label, and the script does not exclude them. (3) It removes the families that have less than 20 apps. Since these details are not mentioned in the paper, and are not found in the other date-scripts either, we have removed the apps that are mapped to MALICIOUS and "none" labels and we have kept all the apps we were able to identify families for.

[17] http://chapering.github.io/droidfax/

[18] https://bitbucket.org/haipeng_cai/duaf/src/master/duafdroid/

Table 12 Performance of DroidCat for malware detection and family identification

	Malware detection						Family identification					
	Original results			Our results			Original results			Our results		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
D1617	99.31%	99.27%	99.28%	86.27%	79.32%	81.23%	94.79%	94.74%	94.54%	51.21%	49.19%	47.23%
D1415	97.26%	97.09%	97.16%	79.47%	80.22%	79.00%	97.84%	97.75%	97.70%	27.34%	32.02%	27.78%
D1213	96.38%	96.04%	96.12%	80.90%	79.56%	79.65%	99.73%	99.71%	99.70%	52.95%	51.81%	50.38%
D0911	97.19%	96.96%	97.00%	79.13%	78.98%	79.05%	99.48%	99.43%	99.44%	39.94%	52.43%	42.93%
mean	97.53%	97.34%	97.39%	81.44%	79.52%	79.73%	97.96%	97.91%	97.84%	42.86%	46.36%	42.08%
stdev	1.25%	1.37%	1.34%	3.31%	0.52%	1.04%	2.27%	2.28%	2.38%	11.84%	9.66%	10.01%

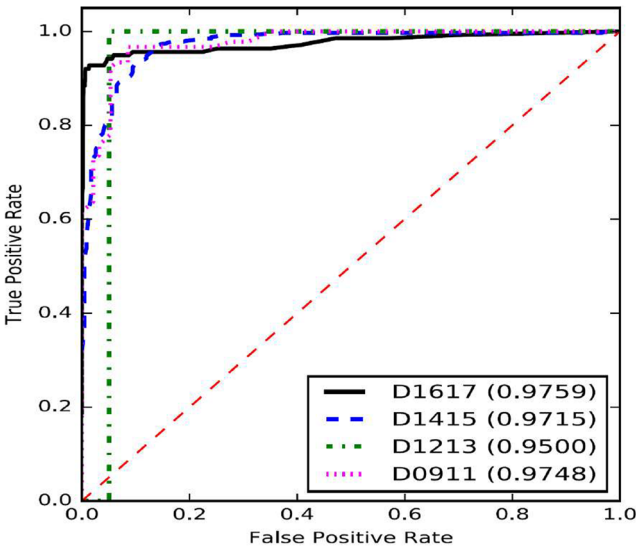

Fig. 6 ROC curve for malware detection from DroidCat publication

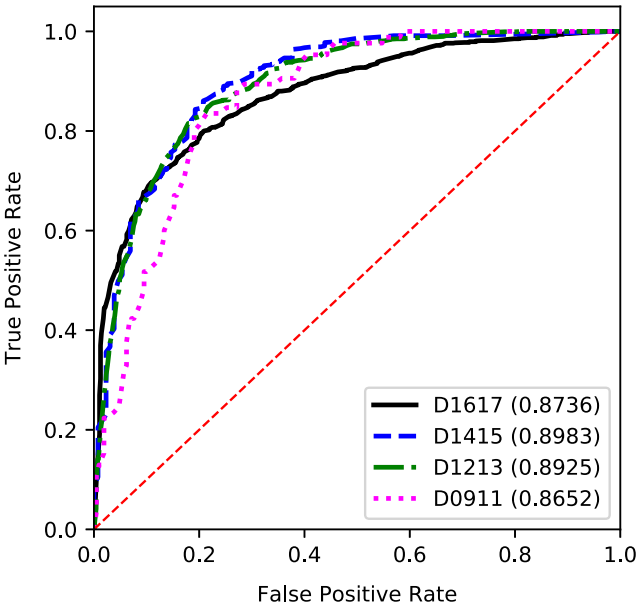

Fig. 7 ROC curve for malware detection from our reproduction

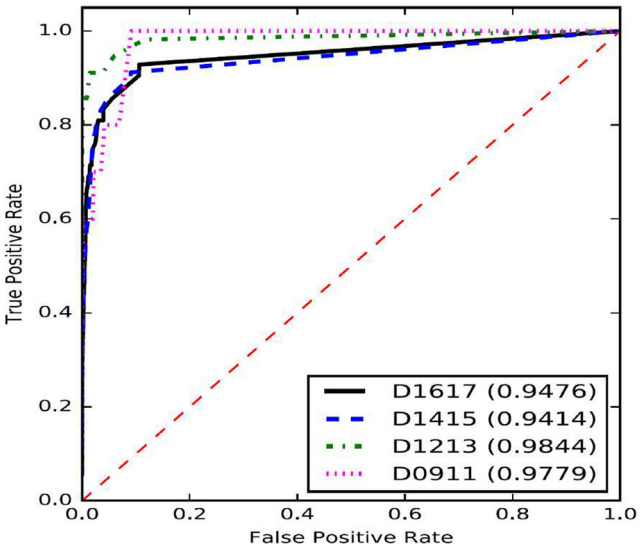

Fig. 8 ROC curve for family detection from DroidCat publication

3.5.4 Results

We have reproduced the experiments that evaluate DroidCat's performance for malware detection as well as malware categorisation. DroidCat's publication reports the results for its two working modes in terms of the weighted average of Precision, Recall, and F1. The ROC curves of the technique is also provided for the four datasets.

Malware detection We present in Table 12 our results compared to original authors results. We notice that our scores for the reproduction experiments are not as good as the scores reported by the DroidCat's publication. Table 12 shows a difference of 16.09 percentage points for mean precision, 17.82 percentage points for mean recall, and 17.66 percentage points for mean F1 (Fig. 6).

The ROC curve of our reproduction is presented in Fig. 7. When compared to the original ROC curve presented in Fig. 6, the two figures enable to draw similar conclusion being unable to reach the same detection performance reported in the DroidCat publication (Fig. 8).

Family identification Our results for family detection compared to DroidCat's publication are presented in Table 12. The reproduction scores are remarkably lower than the original scores on the four datasets. The same observation is made for our ROC curve provided in Fig. 9 when it is compared to DroidCat's family detection ROC curve presented in Fig. 8.

3.6 MalScan

In 2019, MalScan has been presented at ASE as a graph-based static analysis approach that detect Android malware. MalScan considers function call graphs extracted from the apps as social networks and perform centrality analysis to represent the semantic features of the

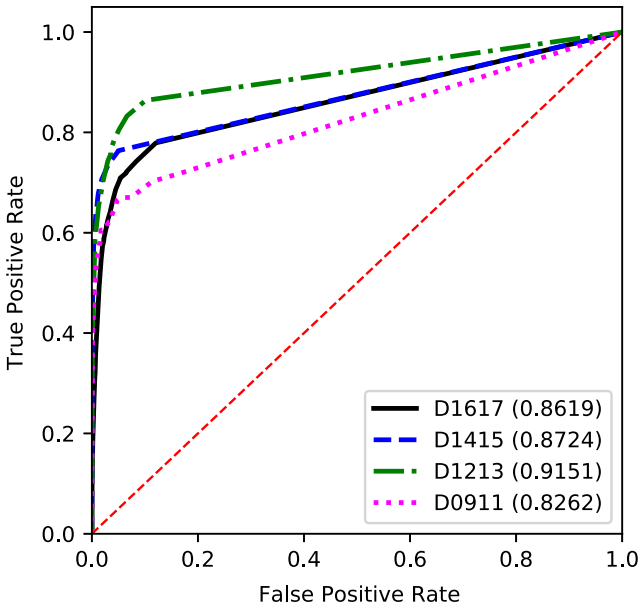

Fig. 9 ROC curve for family detection from our reproduction

graphs. The approach considers six different centrality measures: degree, katz, closeness, harmonic, average, and concatenate.

3.6.1 Dataset

MalScan's dataset contains a total of 30 715 samples that include 15 285 benign apps and 15 430 malicious apps collected from AndroZoo. These apps cover the period from 2011 to 2018, and their hashes are provided by original authors. Based on these lists, we were able to collect all the apps from AndroZoo. We present in Table 13 a summary of this dataset.

3.6.2 Feature Extraction and Embedding

To extract its features, MalScan uses AndroGuard (Desnos and Gueguen 2011) to generate the function call graph that it considers as a social network. The functions and their call relationships in the function call graph are regarded as actors and social interactions in the social network.

Table 13 MalScan dataset

	2011	2012	2013	2014	2015	2016	2017	2018
Benign	1920	1875	1896	1826	1811	2015	1884	2058
Malicious	1916	2000	2000	1982	1839	1940	1834	1919

The social network is used to compute the centrality of sensitive API calls based on PScout's results (Au et al. 2012). As the centrality enables to measure how important a node is within the network, MalScan calculates four centrality measures, each one of them is considered as a set of features. These centrality measures are: degree centrality (Freeman 1978), katz centrality (Katz 1953), closeness centrality (Freeman 1978), and harmonic centrality (Marchiori and Latora 2000). The authors have also constructed two other centrality measures which are: average centrality (the average of the four former centrality measures), and concatenate centrality (the concatenation of the four centrality measures).

The dimension of the feature vectors generated with degree, katz, closeness, harmonic, and average centrality measures is the total number of sensitive API calls from PScout's results which is 21 986 (Au et al. 2012). As for concatenate centrality, the dimension of its feature vectors is 87 944.

The code for the features extraction and vectors embedding is made available by original authors. Using their code, we were able to generate the feature vectors for degree, katz, closeness, and harmonic centrality measures. As of average and concatenate centrality feature vectors, they are not implemented in the code. We had thus to write this part of the code so our reproduction is complete.

3.6.3 Classification

MalScan's authors mention in their paper that they select three algorithms to perform the classification: 1-Nearest Neighbors, 3-Nearest Neighbors, and Random Forest. The evaluation of the approach is based on 10-fold cross-validation, and Random Forest is implemented with its default parameters as cited by the authors. The code for this part is also provided by the authors and it enables to compute F1 score and Accuracy for the three ML algorithms.

As we reproduced the experiments that assess the detection effectiveness of MalScan, we were unable to decide about the exact ML algorithm used to report the results presented in Section IV-B Table V of the original MalScan paper. The authors do not explicitly name the algorithm they have chosen for the detection effectiveness. As a workaround, we have used 1-Nearest Neighbor to compare with their results. We base our choice on the affirmation of the authors in section IV-D which assesses the Robustness of MalScan against Adversarial Attacks. In this same section, the authors affirm that they select 1-NN for this experiment since it provides better effectiveness on malware detection. Nonetheless, we are not sure whether it is the algorithm used to report the results in section IV-B Table V.

3.6.4 Results

As mentioned in the previous section, we have reproduced the experiments that evaluate the detection effectiveness of MalScan. The results are reported in terms of Accuracy and F1 for the eight datasets using the six sets of features. In total, 48 experiments are reproduced. We report our metrics in Table 14 and the original results in Table 15.

From the two Tables, we can notice that both F1 score and Accuracy of the original paper are higher in 30 out of 48 experiments (with a largest difference of 1.5 percentage points), smaller in 12 experiments, and equal to our results in 6 experiment. Overall, our reproduction results are very similar to the results reported in the original publication.

Table 14 Detection effectiveness of MalScan from our reproduction

Dataset	2011		2012		2013		2014		2015		2016		2017		2018	
Metrics	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A
Degree	94.7	94.6	96.4	96.2	96.4	96.2	95.6	95.4	98.2	98.2	98.4	98.5	96.7	96.7	97.6	97.7
Closeness	96.1	96.0	96.9	96.7	96.5	96.4	96.8	96.6	97.7	97.6	97.5	97.5	97.4	97.4	98.3	98.4
Harmonic	97.1	97.1	97.9	97.9	97.1	97.0	96.5	96.3	96.0	96.0	97.4	97.4	97.0	97.1	97.6	97.7
Katz	96.0	95.9	96.4	96.2	97.1	97.0	96.8	96.7	97.2	97.2	97.4	97.5	98.2	98.2	98.0	98.1
Average	97.0	97.0	98.0	97.9	97.1	97.0	96.5	96.3	95.8	95.8	97.4	97.4	97.1	97.1	97.9	97.9
Concatenate	97.1	97.1	97.9	97.9	97.1	97.0	96.5	96.3	96.1	96.1	97.4	97.4	97.0	97.0	97.8	97.9

Table 15 Detection effectiveness of MalScan from the original publication

Dataset	2011		2012		2013		2014		2015		2016		2017		2018	
Metrics	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A
Degree	95.5	95.4	96.7	96.5	96.4	96.2	95.6	96.4	98.1	98.1	98.5	98.5	97.0	97.0	97.9	98.0
Closeness	96.2	96.1	96.5	96.3	96.7	96.5	96.7	96.5	97.6	97.6	97.3	97.3	97.7	97.7	98.8	98.8
Harmonic	96.9	96.8	98.0	97.9	97.2	97.1	96.8	96.6	96.0	96.1	97.2	97.3	96.8	96.8	97.9	98.0
Katz	96.0	95.8	96.4	96.1	96.9	96.8	96.6	96.5	97.4	97.4	98.0	98.0	98.4	98.4	98.0	98.1
Average	97.2	97.2	97.9	97.9	97.2	97.1	96.6	96.5	96.7	96.7	97.7	97.7	96.9	97.0	98.3	98.3
Concatenate	97.5	97.5	98.1	98.0	97.5	97.4	97.7	97.6	97.6	97.6	97.7	97.7	97.1	97.1	98.4	98.5

4 Lessons Learnt

Reproducing the experimental setups, and hence results, from the reproduced approaches takes a significant effort. In Section 3, while describing the reproduction steps, we enumerated the challenges that we face to pinpoint all relevant information, and detailed the guesswork that was necessary to pursue the reproduction attempts. In Section 4,

(1) we present the outcomes of our reproduction attempt and, in order to allow readers to fully grasp the realities of those often overlooked aspects; (2) we offer illustrations of concrete difficulties that we faced; (3) we conclude on the divergence of reproduction results.

4.1 Outcome of our Reproduction Attempts

After putting extensive effort to reproduce DREBIN, MaMaDroid, RevealDroid, DroidCat, and MalScan, and based on the experimental results that we compare against the performance presented by the authors in their publications, we formulate the following question:

Can our reproduction attempt be considered as successful?

For DREBIN, MaMaDroid, RevealDroid, and DroidCat, our **reproduction** attempt could not achieve 100% identical experimental setup. Missing data, software artefacts and/or details resulted in deviations from the original setups—the causes and consequences of those deviations are discussed below. Following the precise terminology defined in Introduction, we come to the conclusion that we have failed to **reproduce** DREBIN, MaMaDroid, RevealDroid, and DroidCat.

As for MalScan we have been able to report similar results using the same exact experimental setup provided by original authors. Following the same ACM terminology, we can conclude that we have succeeded in **reproducing** the MalScan approach.

> **Finding 1**: One out of five approaches can be **reproduced**.

For DREBIN, MaMaDroid, and RevealDroid, We have nonetheless achieved a **replication** that we strove to keep as *close* as possible to the original approaches (e.g., by replacing missing dataset samples with samples from the same sources and from the same period of collection).

The three replications yielded *similar* results to the original approaches. The performance differences could very well be explained by the deviations of our experimental setups. Hence, we have to conclude that the **replication** of DREBIN MaMaDroid, and RevealDroid is successful, i.e., we were able to achieve performance that is very close to what original authors reported. However, an important (if not huge) engineering effort has been required to obtain a fair replicability of these approaches.

> **Finding 2**: Three out of four approaches can be **replicated**, but with deviations to the original works.

> **Finding 3**: **Replicating** an approach requires a significant effort from other researchers.

As for DroidCat, we have failed both to reproduce and to replicate the approach. The results of our experiments are not even *fairly similar* to original results. Our failure can be explained by the difference of our dataset. The effect of this difference is more significant on the familial classification, since the labels are determined based on malware apps of our dataset. Moreover, we were not sure if some of the scripts we have used are the same scripts used to report original results. Consequently, we have ended up with a different experimental setup.

Finding 4: One out of five approaches could neither be **reproduced** nor **replicated**

4.2 Insights from our Reproduction Journey

4.2.1 Dataset Re-acquisition is Hard

The first category of problems that we have identified are barriers to re-acquire the dataset used to yield published results.

While we were able to obtain the full dataset for our MalScan reproduction attempt, for DREBIN, MaMaDroid, RevealDroid, and DroidCat approaches, we were unable to obtain 100% identical datasets. When it was possible, We resorted to retrieve dataset samples that are as *similar* as possible to the original datasets. Even so, this task proved to require a significant amount of work and time.

- DREBIN: As mentioned before, we have succeeded to collect only 46,42% of the same original goodware APKs. Completing the goodware dataset has turned out to be necessary to simulate the original experimental setup. Our goodware dataset is composed of samples selected from the same time period as DREBIN original goodware dataset. However, these samples do not present the same exact APK files.
- MaMaDroid: The authors do not directly provide the APK files but the list of hashes, although they indicated the original source of the apps (DREBIN, PlayDrone and VirusShare). We have been able to collect from VirusShare almost all necessary apps. We present in Table 17 in Appendix, an illustration of a problem encountered during the collection of the dataset.
- RevealDroid: The authors do not provide the APK files. We had to search in their code repositories for the potential lists of hashes that have been used to report their results. For VirusShare dataset, we have been able to collect the exact same number of apps used in original experiments. As for the other datasets, we have failed to identify and collect the exact same apps. We provide in Table 9 a detailed explanation of the process of collection as well as the problems we have encountered.
- DroidCat: The authors do not provide the APK files, but the lists of the hashes. However, we have noticed a significant mismatch between the number of apps in the original publication and the number in the lists. We have again failed to reproduce the exact same dataset.

Finding 5: Obtaining datasets is challenging. Unless original authors ensure their dataset is available, **Reproduction** is doomed to failure.

4.2.2 Feature Extraction Process can be Puzzling

Once a raw dataset is collected, the next step is to extract features. For MalScan, this process has been straightforward using original code. As for the other approaches, this extraction has brought its own barriers to reproduction.

- DREBIN: Authors do not provide the code for extracting the features. Thus, we have used a re-implementation of DREBIN's code that is made publicly available by other authors[19] (Narayanan et al. 2017).
- MaMaDroid: The code is made publicly available by original authors. However, we have faced problems to generate the call graphs from which features are extracted. In particular:
  - We noticed that none of the provided versions of the software exactly matched the description of any of the three versions of the paper. Similarly, the code depends on data files for FlowDroid that users are expected to obtain from the FlowDroid repository, but no mention is made of the exact version of those files.
  - A specific directory structure is supposed to be respected for the process to succeed. Nonetheless, neither this structure nor the expected output are documented in the code. We note that if this directory structure is not respected, an intermediate output file can still be generated, but it does *not* represent the expected call graph that can be used to create the vectors space.

    In Table 17 in Appendix, we provide an illustration of some of the encountered problems.
- RevealDroid: The authors provide their code, but still we were somehow confused about the exact script we need to use to extract API usage features and their outputs.
- DroidCat: The code is made available by original authors. However, the features extractor crashed on several apps due to Soot not being able to instrument the apps.

Finding 6: Even when the code is provided, **Reproduction** and/or **Replication** can require massive guesswork.

4.2.3 Vector Embedding Code may be Buggy

After overcoming several challenges related to feature extraction, embedding the extracted features into a vector space comes with its share of problems.

- DREBIN: Original code is not available. However, the process appeared to be well documented in the publication to enable reproduction of the feature embedding procedure.
- MaMaDroid: Authors make their code publicly available. However, the API version level of the list of the known packages provided in the code (28 or 26) differs from the version stated in the original paper (24). This mismatch has made a successful reproduction task even more difficult. An illustration of the encountered problems is presented in Table 17 in Appendix. This case raises another important question in the context of a reproduction: Should the code be run "as is", even though it demonstrably differs from the explanation provided in the original paper?

[19] https://github.com/MLDroid/drebin

- RevealDroid: The authors make their code repositories publicly available. However, since the vector embedding part is not included in their repositories' documentation, we were unable to identify the script that is supposed to create the features vectors among the variety of files they provide.
- DroidCat: With the original code provided, we have faced some issues in computing the features. These issues are caused by the author's tool duafdroid that has failed to compute the features on several apps.
- MalScan: The authors share their code that generates the features vectors of four out of six sets of features. For the missing vectors, we had to write our own implementation based on the details provided in the original publication.

Finding 7: Even when the code is provided, **Reproduction** and/or **Replication** can involve significant code rework

4.2.4 Classification Procedure may Require Clarifications

- DREBIN: The code again is not available. We have then relied on externally re-implemented code[20] (Narayanan et al. 2017), and still, we had to recover ourselves several undisclosed, though crucial, hyper-parameters such as C. Original authors have not documented the value used to perform the experiment. We have then decided to use the default value $C = 1$ of scikit-learn framework we use for our experiment. We note that the exact implementation used in original work is also not specified in the paper.
- MaMaDroid: Unlike Features Extraction and Vector Embedding steps, the code that performs the classification is not made available by authors. However, the configuration of Random Forest algorithms is well detailed, and authors provide hyper-parameters values for both family and package modes. We note that the Random Forest implementation used in original work is also not specified in the paper. Thus, we have used the same framework scikit-learn for both reproduced approaches.
- RevealDroid: The code is made available by original authors. But since it requires as input an HDF file, we have combined it with our script that creates the features vectors to avoid storage issues as we have mentioned previously.
- DroidCat: Original authors share their classification scripts. However, we were unable to identify the exact file that has been used to generate original results as mentioned previously. We have also faced problems related to the incompatibility of the vectors space and the input of the classification selected script.
- MalScan: The code is made available by original authors and it calculates the performance metrics of three ML algorithms. However, we were confused about the exact algorithm that is used to report original results since it is not explicitly stated in the paper.

4.3 Conclusions on the Divergence of Reproduction Results

Our reproduction efforts allowed us to obtain performance metrics, that we reported in the same way as in the original works, i.e., mainly (1) a table for MaMaDroid, RevealDroid, DroidCat, and MalScan; (2) a ROC curve for both DREBIN and DroidCat; (3) a PR curve

[20] https://github.com/MLDroid/drebin, that we had to update

for RevealDroid. While there exist methods to average several ROC curves into one single summary ROC curve (Chen and Samuelson 2014), it is unclear whether DREBIN's ROC curve is the result of such a process, or the standalone ROC curve of one specific experiment. As for DroidCat's ROC curves and RevealDroid's PR curves, they are generated using original authors' scripts. For MaMaDroid, RevealDroid, and MalScan, the results provided are averaged over the 10-fold cross-validation splits. None of the five reproduced approaches provided a complete *distribution* of results, that would enable us to perform a statistical comparison between their results and ours. As a consequence, we have to conclude whether our results confirm those of the original authors based on single data points comparison. For instance, we had to compare DroidCat's Figs. 6 and 7 and decide, somewhat subjectively whether *we* consider them to be "similar enough". Similarly, when comparing the F-measure for MaMaDroid's experiment using Package mode (PCA) and `2016, newbenign` and our replication, we were left to our own judgment to decide whether 0.85 is "similar enough" to 0.89.

To the best of our knowledge, there is no universally accepted method to objectively decide when a single data point difference must be considered *abnormal*.

Conclusion 1: There is no systematic and definitive method for deciding the success or failure of a **Reproduction** and/or **Replication** attempt.

This problem is exacerbated by the differences of our reproduction attempts against the original works. Indeed, in order for experimenters to be able to conclude that a replication reasonably proves that the claims of the original paper are wrong, it would be necessary to prove that the performance mismatch cannot possibly be explained by the differences of the reproduction.

Conclusion 2: In practice, unless they are fully reproducible, machine learning based Android Malware detection approaches are **unfalsifiable** (i.e., they do not leave room for **refutability** checks).

5 Reproduction in the Bigger Picture

Our first intention for this work was to perform a complete reproduction of the five approaches. The availability of the artefacts and the similarity of our results to original publication has made us conclude that we have successfully reproduced MalScan approach. However, some missing artefacts and description details have made the reproduction of the four other approaches impossible, and our attempt has thus turned into a replication exercise. The results reported with Drebin, MaMaDroid, and RevealDroid reproduction/replication attempt can confirm that we have fairly similar results as those stated in the original publications after investing a substantial amount of effort to re-engineer the process and fill the datasets so that they are as close as possible to the literature experiments. As for DroidCat, the huge difference between our reproduction/replication results and original publication has made us conclude that we have failed both to reproduce and to replicate the approach.

We described all the difficulties that we have faced, starting from the missing APK files, to identifying and fixing bugs in the partial code that was released by authors. Our reproduction/replication of some approaches has put us in puzzling situations namely when the information in the code did not match what is stated in the paper. This mismatch was explicit for differences between the API version levels of MaMaDroid approach for instance.

The mismatch was however harder to detect for the case where the code was buggy. In this second case we have discovered some parts of the code that do not perform what is described in the papers.

We understand, through our own experience, that releasing a reproducible approach is not a straightforward exercise. It is however, still, very necessary to advance the research. With the MalScan approach, we can learn that releasing a reproducible approach is possible. Reproduction works are essential as they enable, on the one hand, to validate the approach, and on the other hand, to leverage existing work, and to build on it to advance the research domain. The difficulty of reproduction has also been addressed in other research fields and has shown that researchers may not just fail to reproduce other researchers' works, but also to reproduce their own results. The repeated unsuccessful reproduction attempts has led to what is called the *Reproduction Crisis*.

Maturity of the research field The use of Machine learning for malware detection has been investigated for many years in the Android research community. Several approaches have indeed been presented at second tier venues. Nevertheless, our study about the subjects selection has showed that in the last ten years, 24 out of 26 806 papers published at 16 top venues deal with android malware detection. On the one hand, the scarcity of research results at top-tier venues questions the significance of the achievements made so far by the community. On the other hand, literature approaches continuously report high performance scores with different approaches.

Our failed reproduction attempts for four out of five approaches now stresses on the following question that the community should answer:

> *"Is machine learning based Android malware detection a mature research field?"*

With TESSERACT (Pendlebury et al. 2019), the authors have recently raised concerns of limitations and blind spots in experimental evaluations, following up on previous studies by Allix et al. (2016a) and Allix et al. (2015). These studies however did not consider the deeper question of reproducibility. Thus, despite 10 years of tentative approaches and published record performance measurements, it remains unclear in which direction the research field is moving. In short, one wonders:

> *"What does the community know now on the success of machine learning-based malware detection that we did not know with the early works?"*

In practice, no comparison seems to be actually feasible among the state of the art. The target working scenario/context of each approach is virtually never defined, due to imprecisions in approach design, non-detailed descriptions of experimental protocols, lack of agreed upon Benchmarks, lack of artefacts, and, more generally, *non-reproducibility* of experiments.

Theoretical foundations State-of-the-art machine learning based approaches for malware detection such as MaMaDroid and Drebin are presented as a careful orchestration of design choices leading to a performance feat that deserves publication. Unfortunately, the evaluation of such approaches generally eludes an assessment of the added value. Typically, it is unclear which steps in the learning pipeline is the key differentiator in the Drebin and MaMaDroid approaches in terms of performance impact. One could question indeed whether the feature set or the ML algorithm, among many other parameters, could be, for example, the main contributors to the performance. Overall, a concern that can be raised is the lack of justification to the different parameter choices (including not only learning hyper-parameters, but also algorithm choice and feature engineering). In short the published content often overlooks a large part of the story behind the approach development and propose whole new approach instead of leveraging previous advances. *Reproducibility* of the experiments, however, could have resolved such issues by allowing other researchers to systematically experiment and introduce variations in the different steps. This, furthermore, might have resulted in an overall *move forward* of the field where new approaches build on previous ones, instead of the plethora of seemingly unconnected approaches the domain has seen. To summarise, the failed reproducibility attempts raise the following questions:

> *"Has the literature offered any hindsight on what works? Has the field grown any wisdom, or any testable hypothesis on what might work?"*

On the need for a transparent experimental framework During our reproduction journey, we have stumbled upon several obstacles, which other research fields dealing with the use of machine learning are also facing. A recurrent challenge is that the experimental pipelines implemented in current research papers are heavily ad-hoc. A solution to the reproduction problem could be to enforce the use of standardised machine learning pipeline management tooling, with a strong separation of concerns. Ideally, a new proposed approach would be fully characterised by a configuration file describing all the technical parameters. This would eliminate uncertainty, guarantee reproducibility, and allow researchers to investigate the effect of each parameter and to rigorously compare different approaches.

Industry is facing similar problems in the management of the full lifecycle of Machine-Learning systems. While it is unrealistic to expect researchers to deliver production-grade systems, the tooling developed to address production issues, such as TensorFlow Extended[21] or Metaflow[22] could also be used by researchers to make their approaches more re-usable.

6 Related Work

Our work is related to various research directions in the literature. With respect to our reproduction subjects, a number of research works have attempted to compare against them (cf. Section 6.1). Other researchers have already presented investigation results on the biases in the machine learning-based malware detection field (cf. Section 6.2). We also discuss how reproduction is viewed in the scientific literature beyond the research community

[21] https://www.tensorflow.org/tfx
[22] https://metaflow.org/

(cf. Section 6.3). Finally, we enumerate a few strategies proposed in other fields towards ensuring reproducibility (cf. Section 6.4).

6.1 Comparison Against the Reproduced Approaches in the Litterature

DREBIN has been published in 2014, and it is considered as a state of the art approach in android malware detection. Thus, it has rapidly become the center of many related works. To demonstrate the capability of their CASANDRA online learning based framework for malware detection, (Narayanan et al. 2017) have provided an experimental comparison with the DREBIN (as well as with the work of (Allix et al. 2016a)). As part of their experimental setup, they have used DREBIN's malware dataset with 5000 randomly chosen goodware apps from the same period of DREBIN. While they used the same malware dataset, the same features, and the same algorithm as DREBIN, they did not attempt to reproduce DREBIN. Thus, they developed their own experimental setup for comparison.

This process is performed over and over in the literature (Abaid et al. 2017; Demontis et al. 2019; Jordaney et al. 2016; Chen et al. 2019) where authors attempt to obtain the training datasets and implement feature extraction of DREBIN based on the descriptions in the paper. Although authors consider these as reproductions, comparing the obtained results with the original works, which may have different parameters, is not sound.

DREBIN and MaMaDroid are also often reused to perform approach assessments and comparisons. For example, Skovoroda and Gamayunov (2017) have proposed a heuristic approach to static analysis of Android applications using 2 collections of malware datasets: DREBIN and ISCX (Abdul Kadir et al. 2015). Chen et al. (2018) have presented KUAFUDET which is a learning enhancing defense system with adversarial detection, and they have applied their poisoning attack to DREBIN, MaMaDroid, and DroidAPIMiner (Aafer et al. 2013). (Smutz and Stavrou 2016) have proposed a method to identify the observations on which the performance of an ensemble classifier is low. They have evaluated their method using PDFrate, which is a PDF malware detector, and they have replicated DREBIN experimental datasets using 100 000 benign and 5000 malicious applications.

Regarding the two recent approaches RevealDroid and DroidCat, they have also been used by researchers to perform comparison. For instance, Scalas et al. (2019) have compared System API-based strategies with RevealDroid and three other approaches. Cai (2020) have also compared the sustainability of their tool DroidSpan against RevealDroid as well as four other systems from the literature. (Pei et al. 2020) have evaluated their approach against eight state of the art techniques that include DREBIN, MaMaDroid, and DroidCat. Also, (Gong et al. 2020) have compared their tool APICHECKER with DREBIN, DroidCat, and nine other approaches from the literature.

It is worth noting that some of the reproduced approaches are evaluated against each other. For example, RevealDroid's authors have compared against re-implementation of DREBIN by selecting subset of features, and MalScan's effectiveness is compared against DREBIN and MaMaDroid approaches.

6.2 Biases in Machine Learning based Malware Detection

In malware research, Rossow et al. (2012) have conducted a survey on 36 malware execution's papers from 2006-2011 to assess the methodological rigour and prudence, and they have highlighted some deficiencies such as the experimental setup which is often not adequately described in the papers.

In 2015, Allix et al. (2015) showed the significant impact of time coherence in the construction of datasets for Android malware detection. They discovered that not ensuring that the training set contains only apps anterior to the test set led to artificially high reported performance of malware detectors.

Allix et al. (2016a) presented an experimental study demonstrating that a popular evaluation methodology (10-fold cross-validation) could also bring seemingly high performance, despite poor performance in a real world scenario.

Pendlebury et al. (2019) published a study measuring how DREBIN and MaMaDroid were affected by the two biases described in both Allix et al. papers introduced above. They built and released TESSERACT, a tool to automatically evaluate those biases in any machine learning-based Android malware detector. TESSERACT requires a trained machine-learning model. This point highlights the need for relevant research approaches to either provide such a trained model, and/or to allow other researchers to build their own. The latter can only be achieved if the approach is reproducible, or at least replicable.

6.3 Reproduction in Research

In a survey published in *Nature* in 2016 and conducted over a pool of 1576 researchers, it has been shown that more than 70% of the respondents had not succeeded in reproducing experiments of another scientist, and more than 50% have been unable to reproduce their own experiments (Baker 2016).

The bombastic number of researches that are published every year and reporting very good results, has raised the question of quality and reproducibility of the research, and has drawn the attention of researchers in the other scientific fields.

Nuijten (2019) has addressed this problem in the field of psychology. They have affirmed that publication bias and human biases are among the main causes of the reproducibility crisis that is triggered by repeated failures of reproduction.

In medical research, a survey has been carried out on faculty members and trainees at MD Anderson Cancer Center and has revealed that 50% of the participants have failed at least one time to reproduce published data (Mobley et al. 2013).

In economics, McCullough et al. (2006) have examined the reproducibility of articles from the online archive "the Journal of Money, Credit, and Banking", which requires the authors to provide the code and the data to make their research reproducible. The results of this work reported that fewer than 15 of over 150 empirical articles could be reproduced.

King (1995) has affirmed that sufficient information for reproduction is usually unavailable in political science, which makes the reproduction often impossible even with the help of the authors.

Reaves et al. (2016) have categorised papers about android application analysis tools from more than 17 top venues based on, among others, the availability of the tools. After evaluation on a representative sample, they have concluded that tools suffer from important issues.

6.4 Improving Reproducibility

To ensure the healthy development of research, the need of reproduction is becoming crucial. Reproduction helps to verify the correctness of the results as well as to improve them. This has motivated researcher to propose solution to improve Reproducibility.

Nuijten (2019) has proposed strategies to be adopted by researchers in order to increase reproducibility in psychology. They have identified four key factors: Improving statistical inference, Pre-registration, Multisite collaborations, and transparency.

In political science, King (1995) has presented *Reproduction Standards*, which requires the existence of sufficient information to understand, evaluate, and build upon prior work without the help of the authors. To encourage adherence to the reproduction standards, they have proposed solutions to be implemented by authors, Tenure and Promotion Review Committees, Editors and Reviewers of Books and Journals, as well as the implementation of Graduate Programs.

In 2015, an openly-peer-reviewed journal "ReScience" was created [23]. It aims to publish explicit reproductions of research under the condition that they have to be open-source and reproducible (Rougier et al. 2017).

The journal Public Finance Review has called for papers that reproduce previous research whether the result of reproduction is positive or negative (Burman et al. 2010).

In malware research, Rossow et al. (2012) have defined guidelines that are crucial for prudent malware experiments. These guidelines are inspired from 4 goals: *transparency*, *realism*, *correctness*, and *safety*.

Several Computer Science conferences now offer a way for research artefacts to be evaluated. Among such conferences are Usenix Security[24], ICSE[25], POPL[26], ACSAC[27],

7 Conclusion

In this paper, we discuss the findings of a reproduction attempt that we made for five approaches on machine learning based malware detection which have had a large influence on the research line. With this work, we make an urgent call to android malware detection researchers to take lessons from our reproduction experience, and to make their approaches and experiments reproducible to avoid, in the near future, a setback due to a *Reproduction Crisis* in our field. We also invite other researchers to invest in reproduction studies, and publishers to encourage and promote this kind of publications.

Our investigation of the related work also revealed that the reproduction problem is not specific to the Computer Science domain. Instead, it seems every branch of scientific inquiry faces its very own *reproduction crisis*. Fortunately, all fields seem equally eager to outgrow this crisis, albeit there are fields that acknowledged the issue and started their correcting efforts long before others.

Our reproduction attempt has been successful for one out of five approaches. As for the other detectors, our reproduction has turned to be a replication study. Somewhat worrying is that our reproduced/replicated approaches are not exactly equivalent to the original approaches. We had to guess several elements, hence introducing potential differences. We also had to modify the artefacts provided by original authors, therefore introducing actual differences. Combined, those differences make it almost impossible to use reproduced/replicated approaches to draw any conclusion on the original approaches. Whatever

[23] https://github.com/ReScience/

[24] https://www.usenix.org/conference/usenixsecurity20/artifact-evaluation-information

[25] https://2019.icse-conferences.org/track/icse-2019-Artifact-Evaluation

[26] https://popl19.sigplan.org/track/POPL-2019-Artifact-Evaluation

[27] https://www.acsac.org/2019/submissions/papers/artifacts/

the findings, experimenters would rightly wonder: *Given that the reproduced system is different from the original one, would this finding also hold for the original system?*.

Even comparing the performance of a new approach against a reproduction comes with the same question: *Does it perform better than RevealDroid, or does it perform better than a reproduced RevealDroid?*.

Our work conclusions call for a large and systematic community effort to tackle head-on the reproduction crisis in the Machine Learning-based malware detection field. We have proposed some next step in this direction with the need for implementing a transparent framework to manage experimental setups.

Appendix

Table 16 MaMaDroid's dataset collection process

Dataset	Description
oldbenign	It is the easiest set to retrieve; It was originally collected as part of the PlayDrone dataset created by Viennot et al. in 2014, and has since been added to *The Internet Archive* project[a] . MaMaDroid authors provide a direct link to the ZIP file that contains the 5879 applications. While we were able to download this file in June 2019, we note that the whole PlayDrone dataset, including this file, is not available anymore at the time of writing (November 2019)[b]
newbenign	For the `newbenign` dataset, the number described in the paper is 2568 apps, but the number provided in the list of hashes is 2555 apps. 1761 apps out of those 2555 were available on AndroZoo (Allix et al. 2016b). We do not have access to any other sources where we could request APKs by their SHA256 hashes, and therefore we were unable to obtain the remaining 794 apps. Hence we had to complete this dataset with applications from AndroZoo, selecting only apps that date back to 7 March 2016, as described in MaMaDroid
drebin	`drebin` dataset is the malware collection used in DREBIN's work, and has been provided by DREBIN's authors as mentioned previously
2013	For `2013`, we were able to obtain the 11 080 files of the list of hashes from VirusShare
2014	For `2014`, we were able to obtain the 24 317 files of the list of hashes from VirusShare
2015	For `2015`, we were able to obtain only 5216 out of the 5314 APKs of the list of hashes from VirusShare. The 98 missing apps were not present in the VirusShare malware collections indicated by authors, nor in any other VirusShare malware collections we had access to
2016	For `2016`, we were able to obtain the 2974 APKs of the list of hashes from VirusShare.

[a] https://archive.org/

[b] *The Internet Archive* website now displays a message "This item is no longer available. Items may be taken down for various reasons, including by decision of the uploader or due to a violation of our Terms of Use." when accessing the link https://archive.org/details/playdrone-apk-e8

Table 17 Problems illustration

Step	Problem illustration
Dataset	VirusShare collects malware of all sorts and for all platforms, hence we had to download 461 GiB of data, only 119 GiB of which were involved in MaMaDroid's experiments. We note that we had to contact VirusShare maintainer to be able to download old *malware sets*, which were hosted on a crashed server. Overall, collecting the required files from VirusShare took more than one month. As for PlayDrone, as noted above, this dataset is not available anymore at the time of writing[a].
Features extraction	While analysing MaMaDroid code, we were puzzled by one detail of the feature extraction process. Once a call graph is obtained, it is represented as a number of strings following the format: `caller_full_name => ['callee1_full_name(param1_type)', 'callee2_full_name(param1_type, param2_type)', [...]]` The way each string representing the list of callees (i.e. the part after "=>") is converted from one string into a list of callees is by spliting the string over the character `','`. However, while callees are indeed separated by `','`, so are parameters for callees that take more than one parameter. As a consequence, the above example would result in a list of three callees: – `callee1_full_name(param1_type)` – `callee2_full_name(param1_type` – `param2_type)` This made us unclear about MaMaDroid authors original intentions. – **Case 1** Authors intentionally want to consider both methods and their parameters. However the rest of the code will discard everything after a `'('`, and hence will fail to consider the first parameter; – **Case 2** Authors do not want to consider the parameters. However, the code will consider all parameters besides the first one. For both scenario, the code contains at least one bug. More importantly, it raises a difficult question: *In the context of a reproduction, should the code be run "as is", or should identified bugs be fixed?*
Vectors embedding	After running MaMaDroid's code, we noticed that the CSV files containing the output of the embedding step, i.e., the feature vectors, have some issues. Almost all the feature vectors have their values set to 0. After examining some applications manually, we effectively noticed that the extracted features were improperly embedded. To understand this issue, we have further examined the code, and we have figured out 2 problems: – For both modes of operation (family and package), authors provide a `txt` file that contains the possible family/package to which the method calls can be abstracted (except obfuscated and self-defined). For family mode, the `txt` file is a list of the following families: `com.google.`, `xml.`, `apache.`, `javax.`, `java.`, `android.`, `dom.`, `json.`, and `dalvik.`. The overall idea of the abstraction in the code is to check if a call starts with one of these families. Let us consider this call as an example: `org.apache.http.conn.ssl.AllowAll HostnameVerifier: void <init>()`, which should be abstracted to `apache` family. However, the script of the authors checks first if this call starts with a family from the previous list, otherwise, it will match it to `obfuscated` or `self defined` families. Since the call starts with `org.apache`, it will not be matched to `apache` family from the list. In order to actually match a call to the apache framework, `apache` needs to be replaced with `org.apache` in the `txt` file that lists the families. A similar issue is noticed with `xml.`, `dom.`, and `json.` families, and we have then replaced them with `org.xml.`, `org.w3c.dom.`, and `org.json.` in the `txt` file of families.

Table 17 (continued)

Step	Problem illustration
	– Besides the issue of the family names that are not complete, we have also noticed that the matching process itself is not performed correctly. In other words, in family mode, the matching script does not properly compare a given call to each element in the `txt file` listing the family names. As a result, in both cases, i.e., either with a list `com.google.`, `org.apache.`, etc. or with a list with short names `google`, `apache`, etc., the abstraction is not performed correctly. A similar issue is noticed with package mode. To overcome these issues, we have first modified the family/package lists in the `txt files` by considering the complete names. Second, we have modified the comparison scripts for both family and package modes of abstraction.

[a]We will share this dataset upon request

Table 18 Papers that do not match the topic

Paper	Explanation
Meng et al. (2019):Securing **Android** App Markets via Modeling and Predicting **Malware** Spread Between Markets, TIFS	It predicts the spread of Android malware between markets
(Sen et al. 2018):Coevolution of **Mobile Malware** and Anti-**Malware**, TIFS	It investigates the use of coevolutionary techniques
(Du et al. 2018):Statistical Estimation of **Malware Detection** Metrics in the Absence of Ground Truth, TIFS	Not about android
(Nissim et al. 2017):ALDOCX: **Detection** of Unknown **Malicious** Microsoft Office Documents Using Designated Active Learning Methods Based on New Structural Feature Extraction Methodology, TIFS	Not about android
Li et al. (2017):Understanding **Android** App Piggybacking: A Systematic Study of **Malicious** Code Grafting, TIFS	An investigation of Android Piggybacked Apps
Xue et al. (2017):Auditing Anti**Malware** Tools by Evolving **Android Malware** and Dynamic Loading Technique, TIFS	A malware generation system
Das et al. (2016):Semantics-Based Online **Malware Detection**: Towards Efficient Real-Time Protection Against **Malware**, TIFS	Not about android
Caviglione et al. (2016):Seeing the Unseen: Revealing **Mobile Malware** Hidden Communications via Energy Consumption and Artificial Intelligence, TIFS	It is about malware exploiting a covert channel
Naval et al. (2015):Employing Program Semantics for **Malware Detection**, TIFS	Not about android
Rastogi et al. (2014):Catch Me If You Can: Evaluating **Android** Anti**Malware** Against Transformation Attacks, TIFS	An evaluation of anti-malware products
Ma et al. (2014):DNSRadar: Outsourcing **Malicious** Domain **Detection** Based on Distributed Cache-Footprints, TIFS	Not about android
O'Kane et al. (2013):SVM Training Phase Reduction Using Dataset Feature Filtering for **Malware Detection**, TIFS	Not about android
Wei et al. (2011):**Malicious** Circuitry **Detection** Using Thermal Conditioning, TIFS	Not about android
Feng et al. (2014): Apposcopy: semantics-based **detection** of **Android malware** through static analysis , FSE	A static analysis approach

Table 18 (continued)

Paper	Explanation
Song and Touili (2013): PoMMaDe: pushdown model-checking for **malware detection** , FSE	Not about android
Chandramohan et al. (2013): A scalable approach for **malware detection** through bounded feature space behavior modeling, ASE	Not about android
Hammad et al. (2018): A large-scale empirical study on the effects of code obfuscations on **Android** apps and anti-**malware** products, ICSE	Evaluation of AV against obfuscation
Rasthofer et al. (2017): Making Malory Behave **Malicious**ly: Targeted Fuzzing of **Android** Execution Environments, ICSE	A framework that generates android execution environment
Tam et al. (2015): CopperDroid: Automatic Reconstruction of **Android Malware** Behaviors, NDSS	It reconstructs behaviors of android malware
Wong and Lie (2016): IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of **Android Malware**, NDSS	It is an android input generator that produces inputs specific to a dynamic analysis tool
Zhou and Jiang (2012): Dissecting **Android Malware**: Characterization and Evolution, S&P	It characterizes Android malware Genome dataset
Xing et al. (2014): Upgrading Your **Android**, Elevating My **Malware**: Privilege Escalation through **Mobile** OS Updating, S&P	It is a service that detects Pileup vulnerabilities
Yan and Yin (2012): DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic **Android Malware** Analysis, Usenix Security	It is a dynamic binary instrumentation tool for Android
Xue et al. (2017): Malton: Towards On-Device Non-Invasive **Mobile Malware** Analysis for ART, Usenix Security	It is a dynamic analysis tool
Neupane et al. (2015): A Multi-Modal Neuro-Physiological Study of Phishing **Detection** and **Malware** Warnings, CCS	Not about android (1 "Android" in ref)
Kolbitsch et al. (2011): The power of procrastination: **detection** and mitigation of execution-stalling **malicious** code, , CCS	"Android" not in pdf text
Xu et al. (2019): BadBluetooth: Breaking **Android** Security Mechanisms via **Malicious** Bluetooth Peripherals, NDSS	A systematic study over the Bluetooth profiles and a Bluetooth validation mechanism
Nappa et al. (2014): CyberProbe: Towards Internet-Scale Active **Detection** of **Malicious** Servers, NDSS	"Android" not in pdf text
Poeplau et al. (2014): Execute This! Analyzing Unsafe and **Malicious** Dynamic Code Loading in **Android** Applications, NDSS	A static analysis tool
Srndic and Laskov (2013): **Detection** of **Malicious** PDF Files Based on Hierarchical Document Structure, NDSS	"Android" not in pdf text
Balzarotti et al. (2010): Efficient **Detection** of Split Personalities in **Malware**, NDSS	"Android" not in pdf text
Pendlebury et al. (2019): TESSERACT: Eliminating Experimental Bias in **Malware Classification** across Space and Time, Usenix Security	Identifying temporal and spatial bias in the Android malware experiments
Jordaney et al. (2017): Transcend: **Detecting** Concept Drift in **Malware Classification** Models, Usenix Security	Identifying concept drift in malware classification experiments
Bayens et al. (2017): See No Evil, Hear No Evil, Feel No Evil, Print No Evil? **Malicious** Fill Patterns **Detection** in Additive Manufacturing, Usenix Security	"Android" not in pdf text

Table 18 (continued)

Paper	Explanation
Wang et al. (2014): Man vs. Machine: Practical Adversarial **Detection** of **Malicious** Crowdsourcing Workers, Usenix Security	An empirical study of adversarial attacks
Kirat et al. (2014): BareCloud: Bare-metal Analysis-based Evasive **Malware Detection**, Usenix Security	"Android" not in pdf text
Kapravelos et al. (2013): Revolver: An Automated Approach to the **Detection** of Evasive Web-based **Malware**, Usenix Security	"Android" not in pdf text
Curtsinger et al. (2011): ZOZZLE: Fast and Precise In-Browser JavaScript **Malware Detection**, Usenix Security	"Android" not in pdf text
Kolbitsch et al. (2009): Effective and Efficient **Malware Detection** at the End Host, Usenix Security	"Android" not in pdf text
Mirzaei et al. (2019): AndrEnsemble: Leveraging API Ensembles to Characterize **Android Malware** Families, ACM Asia CCS	It is about android malware family characterisation
Gao et al. (2018): Software-Defined Firewall: Enabling **Malware** Traffic **Detection** and Programmable Security Control, ACM Asia CCS	Not about android
Meng et al. (2016): Mystique: Evolving **Android Malware** for Auditing Anti-Malware Tools, ACM Asia CCS	A framework that generates malware with specific features
Rahbarinia et al. (2016): Real-Time **Detection** of **Malware** Downloads via Large-Scale URL->File->Machine Graph Mining, ACM Asia CCS	It is specific to malware download events
Wang et al. (2015): JSDC: A Hybrid Approach for JavaScript **Malware Detection** and **Classification**, ACM Asia CCS	It is about JavaScript malware
Yan (2015): Be Sensitive to Your Errors: Chaining Neyman-Pearson Criteria for Automated **Malware Classification**, ACM Asia CCS	Not about android
Zhang et al. (2014a): **Detection** of stealthy **malware** activities with traffic causality and scalable triggering relation discovery, ACM Asia CCS	Not about android
Wüchner et al. (2014): **Malware detection** with quantitative data flow graphs, ACM Asia CCS	Not about android
Maiorca et al. (2013): Looking at the bag is not enough to find the bomb: an evasion of structural methods for **malicious** PDF files **detection**, ACM Asia CCS	Not about android
Rastogi et al. (2013): DroidChameleon: evaluating **Android** anti**malware** against transformation attacks, ACM Asia CCS	Evaluation of anti-malware products
Zhongyang et al. (2013): DroidAlarm: an all-sided static analysis tool for **Android** privilege-escalation **malware**, ACM Asia CCS	It is about capability leaks identification
Xia et al. (2019): Characterizing and **Detecting Malicious** Accounts in Privacy-Centric **Mobile** Social Networks: A Case Study, SIGKDD	Not about android
Fan et al. (2018): Gotcha - Sly **Malware**!: Scorpion A Metagraph2vec Based **Malware Detection** System, SIGKDD	Not about android
Tamersoy et al. (2014): Guilt by association: large scale **malware detection** by mining file-relation graphs, SIGKDD	Not about android

Table 18 (continued)

Paper	Explanation
Kong and Yan (2013): Discriminant **malware** distance learning on structural information for automated **malware classification**, SIGKDD	Not about android
Ye et al. (2011): Combining file content and file relations for cloud based **malware detection**, SIGKDD	Not about android
Ye et al. (2009): Intelligent file scoring system for **malware detection** from the gray list, SIGKDD	Not about android
Pang et al. (2018): Unorganized **Malicious** Attacks **Detection**, NIPS	Not about android
Wang et al. (2019): Heterogeneous Graph Matching Networks for Unknown **Malware Detection**, IJCAI	Not about android

Artefacts Availability We make all artefacts produced in this work publicly available to the community in order to facilitate future work. https://github.com/Trustworthy-Software/Reproduction-of-Android-Malware-detection-approaches

Acknowledgement This work was partially supported (a) by the Fonds National de la Recherche (FNR), Luxembourg, under project CHARACTERIZE C17/IS/11693861, (b) by the University of Luxembourg under the HitDroid grant, (c) by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830892, and (d) by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under project LuxWAyS.

References

Aafer Y, Du W, Yin H (2013) Droidapiminer: Mining api-level features for robust malware detection in android. In: Zia T, Zomaya A, Varadharajan V, Mao M (eds) Security and privacy in communication networks. Springer International Publishing, Cham, pp 86–103

Abaid Z, Kaafar MA, Jha S (2017) Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers. In: 2017 IEEE 16th international symposium on network computing and applications (NCA), pp 1–10. https://doi.org/10.1109/NCA.2017.8171381

Abdul Kadir AF, Stakhanova N, Ghorbani AA (2015) Android botnets: What urls are telling us. In: Qiu M, Xu S, Yung M, Zhang H (eds) Network and system security. Springer International Publishing, Cham, pp 78–91

Allix K, Bissyandé TF, Klein J, Le Traon Y (2015) Are your training datasets yet relevant? In: Piessens F, Caballero J, Bielova N (eds) Engineering Secure Software and Systems. Springer International Publishing, Cham, pp 51–67. https://doi.org/10.1007/978-3-319-15618-7_5

Allix K, Bissyandé TF, Jérome Q, Klein J, State R, Le Traon Y (2016a) Empirical assessment of machine learning-based malware detectors for android. Empir Softw Eng 21(1):183–211. https://doi.org/10.1007/s10664-014-9352-6

Allix K, Bissyandé TF, Klein J, Le Traon Y (2016b) Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th international conference on mining software repositories, ACM, New York, NY, USA, MSR '16, pp 468–471. https://doi.org/10.1145/2901739.2903508

Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K (2014) Drebin: Efficient and explainable detection of android malware in your pocket. In: Proceedings of the ISOC network and distributed system security symposium (NDSS), San Diego, CA

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49(6):259–269

Association for Computer Machinery (2020) Artifact review and badging. https://www.acm.org/publications/policies/artifact-review-and-badging-current, Accessed 30 Oct 2020

Au KWY, Zhou YF, Huang Z, Lie D (2012) Pscout: Analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '12, pp 217–228. https://doi.org/10.1145/2382196.2382222

Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E (2015) Mining apps for abnormal usage of sensitive data. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 426–436. https://doi.org/10.1109/ICSE.2015.61

Baker M (2016) 1,500 scientists lift the lid on reproducibility. Nature 533:452–454

Balzarotti D, Cova M, Karlberger C, Kirda E, Kruegel C, Vigna G (2010) Efficient detection of split personalities in malware. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010, The Internet Society. https://www.ndss-symposium.org/ndss2010/efficient-detection-split-personalities-malware

Bartel A, Klein J, Le Traon Y, Monperrus M (2012) Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN international workshop on state of the art in java program analysis, association for computing machinery, New York, NY, USA, SOAP '12, pp 27–38. https://doi.org/10.1145/2259051.2259056

Bayens C, Le T, Garcia L, Beyah R, Javanmard M, Zonouz S (2017) See no evil, hear no evil, feel no evil, print no evil? malicious fill pattern detection in additive manufacturing. In: Proceedings of the 26th USENIX Conference on Security Symposium, USENIX Association, USA, SEC'17, pp 1181–1198

Breiman L (2001) Random forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324

Burman LE, Reed WR, Alm J (2010) A call for replication studies. Public Finance Rev 38(6):787–793

Cai H (2020) Assessing and improving malware detection sustainability through app evolution studies. ACM Trans Softw Eng Methodol 29(2) https://doi.org/10.1145/3371924

Cai H, Meng N, Ryder B, Yao D (2019) Droidcat: Effective android malware detection and categorization via app-level profiling. IEEE Trans Inform Forens Secur 14(6):1455–1470

Canfora G, Martinelli F, Mercaldo F, Nardone V, Santone A, Visaggio CA (2019) Leila: Formal tool for identifying mobile malicious behaviour. IEEE Trans Softw Eng 45(12):1230–1252

Caviglione L, Gaggero M, Lalande J, Mazurczyk W, Urbański M (2016) Seeing the unseen: Revealing mobile malware hidden communications via energy consumption and artificial intelligence. IEEE Trans Inform Forens Secur 11(4):799–810

Chandramohan M, Tan HBK, Briand LC, Shar LK, Padmanabhuni BM (2013) A scalable approach for malware detection through bounded feature space behavior modeling. In: 2013 28th IEEE/ACM international conference on automated software engineering (ASE), pp 312–322

Chen S, Xue M, Tang Z, Xu L, Zhu H (2016) 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, ACM, New York, NY, USA, ASIA CCS '16, pp 377–388. https://doi.org/10.1145/2897845.2897860

Chen S, Xue M, Fan L, Hao S, Xu L, Zhu H, Li B (2018) Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. Comput Secur 73:326–344. https://doi.org/10.1016/j.cose.2017.11.007. http://www.sciencedirect.com/science/article/pii/S0167404817302444

Chen S, Xue M, Fan L, Ma L, Liu Y, Xu L (2019) How can we craft large-scale android malware? an automated poisoning attack. In: 2019 IEEE 1st international workshop on artificial intelligence for mobile (AI4Mobile), pp 21–24. https://doi.org/10.1109/AI4Mobile.2019.8672691

Chen W, Samuelson FW (2014) The average receiver operating characteristic curve in multireader multicase imaging studies. British J Radiol 87(1040):20140016. https://doi.org/10.1259/bjr.20140016

Curtsinger C, Livshits B, Zorn BG, Seifert C (2011) ZOZZLE: fast and precise in-browser javascript malware detection. In: 20th USENIX security symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings, USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Curtsinger.pdf

Das S, Liu Y, Zhang W, Chandramohan M (2016) Semantics-based online malware detection: Towards efficient real-time protection against malware. IEEE Trans Inform Forens Secur 11(2):289–302

Demontis A, Melis M, Biggio B, Maiorca D, Arp D, Rieck K, Corona I, Giacinto G, Roli F (2019) Yes, machine learning can be more secure! a case study on android malware detection. IEEE Trans Dependable Secure Comput 16(4):711–724. https://doi.org/10.1109/TDSC.2017.2700270

Desnos A, Gueguen G (2011) Android: From reversing to decompilation. Black Hat Abu Dhabi https://media.blackhat.com/bh-ad-11/Desnos/bh-ad-11-DesnosGueguen-Andriod-Reversing_to_Decompilation_WP.pdf

Du P, Sun Z, Chen H, Cho J, Xu S (2018) Statistical estimation of malware detection metrics in the absence of ground truth. IEEE Trans Inform Forens Secur 13(12):2965–2980

Duvendack M, Palmer-Jones RW, Reed WR et al (2015) Replications in economics: A progress report. Econ Journal Watch 12(2):164–191

Fan M, Liu J, Luo X, Chen K, Tian Z, Zheng Q, Liu T (2018) Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans Inform Forens Secur 13(8):1890–1905

Fan M, Luo X, Liu J, Wang M, Nong C, Zheng Q, Liu T (2019) Graph embedding based familial analysis of android malware using unsupervised learning. In: Proceedings of the 41st international conference on software engineering, IEEE Press, ICSE '19, pp 771–782. https://doi.org/10.1109/ICSE.2019.00085

Fan Y, Hou S, Zhang Y, Ye Y, Abdulhayoglu M (2018) Gotcha - sly malware! scorpion a metagraph2vec based malware detection system. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, association for computing machinery, New York, NY, USA, KDD '18, pp 253–262. https://doi.org/10.1145/3219819.3219862

Feng Y, Anand S, Dillig I, Aiken A (2014) Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, association for computing machinery, New York, NY, USA, FSE, vol 2014, pp 576–587. https://doi.org/10.1145/2635868.2635869

Fokkens A, van ErpM, Postma M, Pedersen T, Vossen P, Freire N (2013) Offspring from reproduction problems: What replication failure teaches us. In: Proceedings of the 51st annual meeting of the association for computational linguistics (vol 1: Long Papers), Association for Computational Linguistics, Sofia, Bulgaria, pp 1691–1701. https://www.aclweb.org/anthology/P13-1166

Freeman LC (1978) Centrality in social networks conceptual clarification. Soc Netw 1(3):215–239

Gao S, Li Z, Yao Y, Xiao B, Guo S, Yang Y (2018) Software-defined firewall: Enabling malware traffic detection and programmable security control. In: Proceedings of the 2018 on asia conference on computer and communications security, association for computing machinery, New York, NY, USA, ASIACCS '18, pp 413–424. https://doi.org/10.1145/3196494.3196519

Garcia J, Hammad M, Malek S (2018) Lightweight, obfuscation-resilient detection and family identification of android malware. ACM Trans Softw Eng Methodol 26(3) https://doi.org/10.1145/3162625

Gascon H, Yamaguchi F, Arp D, Rieck K (2013) Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on artificial intelligence and security, ACM, New York, NY, USA, AISec '13, pp 45–54. https://doi.org/10.1145/2517312.2517315

Gong L, Li Z, Qian F, Zhang Z, Chen QA, Qian Z, Lin H, Liu Y (2020) Experiences of landing machine learning onto market-scale mobile malware detection. In: Proceedings of the Fifteenth european conference on computer systems, association for computing machinery, New York, NY, USA, EuroSys '20. https://doi.org/10.1145/3342195.3387530

Gundersen OE, Kjensmo S (2018) State of the art: Reproducibility in artificial intelligence. In: McIlraith S, Weinberger K (eds) Proceedings of the 32nd AAAI conference on artificial intelligence (AAAI-18), association for the advancement of artificial intelligence

Hammad M, Garcia J, Malek S (2018) A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: Proceedings of the 40th international conference on software engineering, association for computing machinery, New York, NY, USA, ICSE '18, pp 421–431. https://doi.org/10.1145/3180155.3180228

Hearst MA, Dumais ST, Osuna E, Platt J, Scholkopf B (1998) Support vector machines. IEEE Intell Syst Appl 13(4):18–28. https://doi.org/10.1109/5254.708428

Hou S, Ye Y, Song Y, Abdulhayoglu M (2017) Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining, association for computing machinery, New York, NY, USA, KDD '17, pp 1507–1515. https://doi.org/10.1145/3097983.3098026

Hou S, Ye Y, Song Y, Abdulhayoglu M (2018) Make evasion harder: An intelligent android malware detection system. In: Proceedings of the twenty-seventh international joint conference on artificial intelligence, IJCAI-18, International joint conferences on artificial intelligence organization, pp 5279–5283. https://doi.org/10.24963/ijcai.2018/737

Hutson M (2018) Artificial intelligence faces reproducibility crisis. Science 359(6377):725–726. https://doi.org/10.1126/science.359.6377.725, https://science.sciencemag.org/content/359/6377/725

Islam R, Henderson P, Gomrokchi M, Precup D (2017) Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. In: Reproducibility in machine learning workshop (ICML). arXiv:1708.04133.pdf

Jerome Q, Allix K, State R, Engel T (2014) Using opcode-sequences to detect malicious android applications. In: 2014 IEEE international conference on communications (ICC), pp 914–919 https://doi.org/10.1109/ICC.2014.6883436

Jordaney R, Wang Z, Papini D, Nouretdinov I, Cavallaro L (2016) Misleading metrics: On evaluating machine learning for malware with confidence

Jordaney R, Sharad K, Dash SK, Wang Z, Papini D, Nouretdinov I, Cavallaro L (2017) Transcend: Detecting concept drift in malware classification models. In: Proceedings of the 26th USENIX conference on security symposium, USENIX Association, USA, SEC'17, pp 625–642

Kapravelos A, Shoshitaishvili Y, Cova M, Kruegel C, Vigna G (2013) Revolver: An automated approach to the detection of evasive web-based malware. In: King ST (ed) Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013, USENIX Association, pp 637–652. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/kapravelos

Katz L (1953) A new status index derived from sociometric analysis. Psychometrika 18(1):39–43

Khatter K, Malik S (2015) Androdata: a tool for static & dynamic feature extraction of android apps. Int J Appl Eng Res 10:98–102

Kim T, Kang B, Rho M, Sezer S, Im EG (2019) A multimodal deep learning method for android malware detection using various features. IEEE Trans Inform Forens Secur 14(3):773–788

King G (1995) Replication, replication. PS: Polit Sci Polit 28(3):444–452. https://doi.org/10.2307/420301

Kirat D, Vigna G, Kruegel C (2014) Barecloud: Bare-metal analysis-based evasive malware detection. In: Proceedings of the 23rd USENIX conference on security symposium, USENIX association, USA, SEC'14, pp 287–301

Kolbitsch C, Comparetti PM, Kruegel C, Kirda E, Zhou X, Wang X (2009) 18th USENIX security symposium, Montreal, Canada, August 10-14, 2009, Proceedings, USENIX Association. In: Monrose F (ed), pp 351–366. http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf

Kolbitsch C, Kirda E, Kruegel C (2011) The power of procrastination: Detection and mitigation of execution-stalling malicious code. In: Proceedings of the 18th ACM conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '11, pp 285–296. https://doi.org/10.1145/2046707.2046740

Kong D, Yan G (2013) Discriminant malware distance learning on structural information for automated malware classification. In: Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining, association for computing machinery, New York, NY, USA, KDD '13, pp 1357–1365. https://doi.org/10.1145/2487575.2488219

Van der Kouwe E, Heiser G, Andriesse D, Bos H, Giuffrida C (2019) Sok: Benchmarking flaws in systems security. In: European Conference on Security and Privacy (EuroS&P). IEEE, Stockholm

Lam P, Bodden E, Lhoták O, Hendren L (2011) The Soot framework for Java program analysis: A retrospective. In: Cetus Users and Compiler Infrastructure Workshop, Galveston Island, TX

Li L, Li D, Bissyandé TF, Klein J, Le Traon Y, Lo D, Cavallaro L (2017) Understanding android app piggybacking: A systematic study of malicious code grafting. IEEE Trans Inform Forens Secur 12(6):1269–1284

Ma X, Zhang J, Tao J, Li J, Tian J, Guan X (2014) Dnsradar: Outsourcing malicious domain detection based on distributed cache-footprints. IEEE Trans Inform Forens Secur 9(11):1906–1921

Maiorca D, Corona I, Giacinto G (2013) Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '13, pp 119–130. https://doi.org/10.1145/2484313.2484327

Marchiori M, Latora V (2000) Harmony in the small-world. Physica A: Stat Mechan Appl 285(3-4):539–546

Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G (2016) Mamadroid: Detecting android malware by building markov chains of behavioral models. arXiv:161204433

Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G (2017) Mamadroid: Detecting android malware by buildin markov chains of behavioral models. In: ISOC Network and Distributed Systems Security Symposiym (NDSS), San Diego, CA

McCullough BD, McGeary KA, Harrison TD (2006) Lessons from the jmcb archive. J Money Credit Bank 38(4):1093–1107

Meng G, Xue Y, Mahinthan C, Narayanan A, Liu Y, Zhang J, Chen T (2016) Mystique: Evolving android malware for auditing anti-malware tools. In: Proceedings of the 11th ACM on asia conference on computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '16, pp 365–376. https://doi.org/10.1145/2897845.2897856

Meng G, Patrick M, Xue Y, Liu Y, Zhang J (2019) Securing android app markets via modeling and predicting malware spread between markets. IEEE Trans Inform Forens Secur 14(7):1944–1959

Mirzaei O, Suarez-Tangil G, de FuentesJM, Tapiador J, Stringhini G (2019). In: Andrensemble: Leveraging api ensembles to characterize android malware families. In: Proceedings of the 2019 ACM asia conference on computer and communications security, association for computing machinery, New York, NY, USA, Asia CCS '19, pp 307–314. https://doi.org/10.1145/3321705.3329854

Mobley A, Linder SK, Braeuer R, Ellis LM, Zwelling L (2013) A survey on data reproducibility in cancer research provides insights into our limited ability to translate findings from the laboratory to the clinic. Plos One 8(5):1–4. https://doi.org/10.1371/journal.pone.0063221

Nappa A, Xu Z, Rafique MZ, Caballero J, Gu G (2014) Cyberprobe: Towards internet-scale active detection of malicious servers. In: 21st annual network and distributed system security symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, The Internet Society. https://www.ndss-symposium.org/ndss2014/cyberprobe-towards-internet-scale-active-detection-malicious-servers

Narayanan A, Chandramohan M, Chen L, Liu Y (2017) Context-aware, adaptive, and scalable android malware detection through online learning. IEEE Trans Emerg Topics Comput Intell 1(3):157–175. https://doi.org/10.1109/TETCI.2017.2699220

Narayanan A, Chandramohan M, Chen L, Liu Y (2018) A multi-view context-aware approach to android malware detection and malicious code localization. Empirical Softw Engg 23(3):1222–1274. https://doi.org/10.1007/s10664-017-9539-8

Naval S, Laxmi V, Rajarajan M, Gaur MS, Conti M (2015) Employing program semantics for malware detection. IEEE Trans Inform Forens Secur 10(12):2591–2604

Neupane A, Rahman ML, Saxena N, Hirshfield L (2015) A multi-modal neuro-physiological study of phishing detection and malware warnings. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '15, pp 479–491. https://doi.org/10.1145/2810103.2813660

Nissim N, Cohen A, Elovici Y (2017) Aldocx: Detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. IEEE Trans Inform Forens Secur 12(3):631–646

Nuijten MB (2019) Practical tools and strategies for researchers to increase replicability. Development Med Child Neurol 61(5):535–539. https://doi.org/10.1111/dmcn.14054

O'Kane P, Sezer S, McLaughlin K, Im EG (2013) Svm training phase reduction using dataset feature filtering for malware detection. IEEE Trans Inform Forens Secur 8(3):500–509

Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G (2019) Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). ACM Trans Priv Secur 22(2):14:1–14:34. https://doi.org/10.1145/3313391

Pang M, Gao W, Tao M, Zhou ZH (2018) Unorganized malicious attacks detection. In: Proceedings of the 32nd international conference on neural information processing systems, Curran Associates Inc., Red Hook, NY, USA, NIPS'18, pp 6976–6985

Pei X, Yu L, Tian S (2020) Amalnet: A deep learning framework based on graph convolutional networks for malware detection. Comput Secur 93:101792. https://doi.org/10.1016/j.cose.2020.101792. http://www.sciencedirect.com/science/article/pii/S0167404820300778

Pendlebury F, Pierazzi F, Jordaney R, Kinder J, Cavallaro L (2019) TESSERACT: Eliminating experimental bias in malware classification across space and time. In: 28th USENIX security symposium (USENIX Security 19), USENIX Association, Santa Clara, CA, pp 729–746. https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury

Plesser HE (2018) Reproducibility vs. replicability: A brief history of a confused terminology. Front Neuroinformat 11:76. https://doi.org/10.3389/fninf.2017.00076

Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G (2014) Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: 21st annual network and distributed system security symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, The Internet Society, https://www.ndss-symposium.org/ndss2014/execute-analyzing-unsafe-and-malicious-dynamic-code-loading-android-applications

Popper KR (2002) The logic of scientific discovery, 2nd edn. Routledge, London. first published in 1959

Rahbarinia B, Balduzzi M, Perdisci R (2016) Real-time detection of malware downloads via large-scale url-¿file-¿machine graph mining. In: Proceedings of the 11th ACM on asia conference on computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '16, pp 783–794. https://doi.org/10.1145/2897845.2897918

Rasthofer S, Arzt S, Triller S, Pradel M (2017) Making malory behave maliciously: Targeted fuzzing of android execution environments. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 300–311

Rastogi V, Chen Y, Jiang X (2013) Droidchameleon: Evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '13, pp 329–334. https://doi.org/10.1145/2484313.2484355

Rastogi V, Chen Y, Jiang X (2014) Catch me if you can: Evaluating android anti-malware against transformation attacks. IEEE Trans Inform Forens Secur 9(1):99–108

Reaves B, Bowers J, Gorski SA III, Anise O, Bobhate R, Cho R, Das H, Hussain S, Karachiwala H, Scaife N, Wright B, Butler K, Enck W, Traynor P (2016) *droid: Assessment and evaluation of android application analysis tools. ACM Comput Surv 49(3):55:1–55:30. https://doi.org/10.1145/2996358

Rossow C, Dietrich CJ, Grier C, Kreibich C, Paxson V, Pohlmann N, Bos H, v Steen M (2012) Prudent practices for designing malware experiments: Status quo and outlook. In: 2012 IEEE symposium on security and privacy, pp 65–79. https://doi.org/10.1109/SP.2012.14

Rougier NP, Hinsen K, Alexandre F, Arildsen T, Barba LA, Benureau FC, Brown CT, De Buyl P, Caglayan O, Davison AP et al (2017) Sustainable computational science: The rescience initiative. PeerJ Computer Science 3:e142

Scalas M, Maiorca D, Mercaldo F, Visaggio CA, Martinelli F, Giacinto G (2019) On the effectiveness of system api-related information for android ransomware detection. Comput Secur 86:168–182

Schmicker R, Breitinger F, Baggili I (2019) Androparse - an android feature extraction framework and dataset. In: Breitinger F, Baggili I (eds) Digital forensics and cyber crime. Springer International Publishing, Cham, pp 66–88

Sen S, Aydogan E, Aysan AI (2018) Coevolution of mobile malware and anti-malware. IEEE Trans Inform Forens Secur 13(10):2563–2574

Skovoroda A, Gamayunov D (2017) Automated static analysis and classification of android malware using permission and api calls models. In: 2017 15th annual conference on privacy, security and trust (PST), pp 243–24309. https://doi.org/10.1109/PST.2017.00036

Smutz C, Stavrou A (2016) When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In: 23rd annual network and distributed system security symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/when-tree-falls-using-diversity-ensemble-classifiers-identify-evasion-malware-detectors.pdf

Song F, Touili T (2013) Pommade: Pushdown model-checking for malware detection. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, association for computing machinery, New York, NY, USA, ESEC/FSE, vol 2013, pp 607–610. https://doi.org/10.1145/2491411.2494599

Srndic N, Laskov P (2013) Detection of malicious PDF files based on hierarchical document structure. In: 20th annual network and distributed system security symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013, The Internet Society. https://www.ndss-symposium.org/ndss2013/detection-malicious-pdf-files-based-hierarchical-document-structure

Sun M, Li X, Lui JCS, Ma RTB, Liang Z (2017) Monet: A user-oriented behavior-based malware variants detection system for android. IEEE Trans Inform Forens Secur 12(5):1103–1112

Tam K, Khan SJ, Fattori A, Cavallaro L (2015) Copperdroid: Automatic reconstruction of android malware behaviors. In: 22nd annual network and distributed system security symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015, The Internet Society. https://www.ndss-symposium.org/ndss2015/copperdroid-automatic-reconstruction-android-malware-behaviors

Tamersoy A, Roundy K, Chau DH (2014) Guilt by association: Large scale malware detection by mining file-relation graphs. In: Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining, association for computing machinery, New York, NY, USA, KDD '14, pp 1524–1533. https://doi.org/10.1145/2623330.2623342

Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot - a java bytecode optimization framework. In: Proceedings of the 1999 conference of the centre for advanced studies on collaborative research, IBM Press, CASCON '99, p 13. http://dl.acm.org/citation.cfm?id=781995.782008

Viennot N, Garcia E, Nieh J (2014) A measurement study of google play. In: ACM SIGMETRICS Performance evaluation review, ACM, vol 42, pp 221–233

Wang G, Wang T, Zhang H, Zhao BY (2014) Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In: Proceedings of the 23rd USENIX conference on security symposium, USENIX Association, USA, SEC'14, pp 239–254

Wang J, Xue Y, Liu Y, Tan TH (2015) Jsdc: A hybrid approach for javascript malware detection and classification. In: Proceedings of the 10th ACM symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '15, pp 109–120. https://doi.org/10.1145/2714576.2714620

Wang S, Yan Q, Chen Z, Yang B, Zhao C, Conti M (2018) Detecting android malware leveraging text semantics of network flows. IEEE Trans Inform Forens Secur 13(5):1096–1109

Wang S, Chen Z, Yu X, Li D, Ni J, Tang LA, Gui J, Li Z, Chen H, Yu PS (2019) Heterogeneous graph matching networks for unknown malware detection. In: Proceedings of the Twenty-Eighth international joint conference on artificial intelligence, IJCAI-19, International Joint Conferences on Artificial Intelligence Organization, pp 3762–3770. https://doi.org/10.24963/ijcai.2019/522

Wang W, Wang X, Feng D, Liu J, Han Z, Zhang X (2014) Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans Inform Forens Secur 9(11):1869–1882

Wei S, Meguerdichian S, Potkonjak M (2011) Malicious circuitry detection using thermal conditioning. IEEE Trans Inform Forens Secur 6(3):1136–1145

Wong MY, Lie D (2016) Intellidroid: A targeted input generator for the dynamic analysis of android malware. In: 23rd annual network and distributed system security symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/intellidroid-targeted-input-generator-dynamic-analysis-android-malware.pdf

Wu Y, Li X, Zou D, Yang W, Zhang X, Jin H (2019) Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 139–150

Wüchner T, Ochoa M, Pretschner A (2014) Malware detection with quantitative data flow graphs. In: Proceedings of the 9th ACM symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '14, pp 271–282. https://doi.org/10.1145/2590296.2590319

Xia Z, Liu C, Gong NZ, Li Q, Cui Y, Song D (2019) Characterizing and detecting malicious accounts in privacy-centric mobile social networks: A case study. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, association for computing machinery, New York, NY, USA, KDD '19, pp 2012–2022. https://doi.org/10.1145/3292500.3330702

Xing L, Pan X, Wang R, Yuan K, Wang X (2014) Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In: 2014 IEEE symposium on security and privacy, pp 393–408. https://doi.org/10.1109/SP.2014.32

Xu F, Diao W, Li Z, Chen J, Zhang K (2019) Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. https://doi.org/10.14722/ndss.2019.23482

Xu K, Li Y, Deng RH (2016) Iccdetector: Icc-based malware detection on android. IEEE Trans Inform Forens Secur 11(6):1252–1264

Xue L, Zhou Y, Chen T, Luo X, Gu G (2017) Malton: Towards on-device non-invasive mobile malware analysis for ART. In: Kirda E, Ristenpart T (eds) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017, USENIX Association, pp 289–306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xue

Xue Y, Meng G, Liu Y, Tan TH, Chen H, Sun J, Zhang J (2017) Auditing anti-malware tools by evolving android malware and dynamic loading technique. IEEE Trans Inform Forens Secur 12(7):1529–1544

Yan G (2015) Be sensitive to your errors: Chaining neyman-pearson criteria for automated malware classification. In: Proceedings of the 10th ACM symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '15, pp 121–132. https://doi.org/10.1145/2714576.2714578

Yan L, Yin H (2012) Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In: Kohno T (ed) Proceedings of the 21th USENIX security symposium, Bellevue, WA, USA, August 8-10, 2012, USENIX Association, pp 569–584. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan

Yang W, Xiao X, Andow B, Li S, Xie T, Enck W (2015) Appcontext: Differentiating malicious and benign mobile app behaviors using context. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 303–313

Yang W, Prasad M, Xie T (2018) Enmobile: Entity-based characterization and analysis of mobile malware. In: 2018 IEEE/ACM 40th international conference on software engineering (ICSE), pp 384–394

Ye Y, Li T, Jiang Q, Han Z, Wan L (2009) Intelligent file scoring system for malware detection from the gray list. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery

and data mining, association for computing machinery, New York, NY, USA, KDD '09, pp 1385–1394. https://doi.org/10.1145/1557019.1557167

Ye Y, Li T, Zhu S, Zhuang W, Tas E, Gupta U, Abdulhayoglu M (2011) Combining file content and file relations for cloud based malware detection. In: Proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining, association for computing machinery, New York, NY, USA, KDD '11, pp 222–230. https://doi.org/10.1145/2020408.2020448

Ye Y, Hou S, Chen L, Lei J, Wan W, Wang J, Xiong Q, Shao F (2019) Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection. In: Proceedings of the twenty-eighth international joint conference on artificial intelligence, IJCAI-19, International Joint Conferences on Artificial Intelligence Organization, pp 4150–4156. https://doi.org/10.24963/ijcai.2019/576

Zhang H, Yao DD, Ramakrishnan N (2014a) Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In: Proceedings of the 9th ACM symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '14, pp 39–50. https://doi.org/10.1145/2590296.2590309

Zhang M, Duan Y, Yin H, Zhao Z (2014b) Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Ahn G, Yung M, Li N (eds) Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, Scottsdale, AZ, USA, November 3-7, 2014, ACM, pp 1105–1116. https://doi.org/10.1145/2660267.2660359

Zhao K, Zhang D, Su X, Li W (2015) Fest: A feature extraction and selection tool for android malware detection. In: 2015 IEEE symposium on computers and communication (ISCC), pp 714–720. https://doi.org/10.1109/ISCC.2015.7405598

Zhongyang Y, Xin Z, Mao B, Xie L (2013) Droidalarm: An all-sided static analysis tool for android privilege-escalation malware. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security, association for computing machinery, New York, NY, USA, ASIA CCS '13, pp 353–358. https://doi.org/10.1145/2484313.2484359

Zhou Y, Jiang X (2012) Dissecting android malware: Characterization and evolution. In: 2012 IEEE symposium on security and privacy, pp 95–109. https://doi.org/10.1109/SP.2012.16

Zhou Y, Wang Z, Zhou W, Jiang X (2012) Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: Proceedings of the 19th Network and Distributed System Security Symposium NDSS, p 2012

Zhu Z, Dumitraundefined T (2016) Featuresmith: Automatically engineering features for malware detection by mining the security literature. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '16, pp 767–778. https://doi.org/10.1145/2976749.2978304