Assessing and Improving Malware Detection Sustainability through App Evolution Studies

HAIPENG CAI, School of Electrical Engineering and Computer Science, Washington State University

Machine learning-based classification dominates current malware detection approaches for Android. However, due to the evolution of both the Android platform and its user apps, existing such techniques are widely limited by their reliance on new malware samples, which may not be timely available, and constant retraining, which are often very costly. As a result, new and emerging malware slips through, as seen from the continued surging of malware in the wild. Thus, a more practical detector needs not only to be accurate on particular datasets but, more critically, to be able to *sustain* its capabilities over time without frequent retraining. In this paper, we propose and study the sustainability problem for learning-based app classifiers. We define sustainability metrics and compare them among five state-of-the-art malware detectors for Android. We further developed *DroidSpan*, a novel classification system based on a new behavioral profile for Android apps that capture sensitive access distribution from lightweight profiling. We evaluated the sustainability of *DroidSpan* versus the five detectors as baselines on longitudinal datasets across the past eight years, which include 13,627 benign apps and 12,755 malware. Through our extensive experiments, we showed that *DroidSpan* significantly outperformed all the baselines in substainability at reasonable costs, by 6–32% for same-period detection and 21–37% for over-time detection. The main *takeaway*, which also explains the superiority of *DroidSpan*, is that the use of features consistently differentiating malware from benign apps over time is essential for sustainable learning-based malware detection, and that these features can be learned from studies on app evolution.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software evolution**.

Additional Key Words and Phrases: Android apps, malware detection, sustainability, evolution

ACM Reference Format:
Haipeng Cai. 2019. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2019), 28 pages. https://doi.org/10.1145/3371924

1 INTRODUCTION

The increasing dominance of Android among mobile computing platforms [49] is accompanied by its share of the vast majority (over 90%) of all mobile malware [1]. In response to the continued surge of malware in different markets of Android applications (known as *apps*), there has been a growing body of defense solutions against malware [25, 57] being proposed. A major defense technique has been app *classification* based on machine learning (ML), which identifies malware by predicting a given app as benign or malicious [10, 28, 32, 38, 52].

Typically an ML-based classification approach works by first training a classifier based on a set of features extracted from labeled sample apps, and then applying the trained classifier to unlabeled apps using the same feature set. Thus, the key step of such approaches is the extraction of an app's

Author's address: Haipeng Cai, haipeng.cai@wsu.edu, School of Electrical Engineering and Computer Science, Washington State University, 355 NE Spokane St., Pullman, WA, 99163.

1049-331X/2019/1-ART1 $15.00
https://doi.org/10.1145/3371924

features that constitute a *behavior profile* for the app. Existing approaches have explored various kinds of features, computed through static [7, 9, 54, 64, 65], dynamic [3, 22], or hybrid [19, 51] app analysis. Mostly, the features are based on apps' usage of permissions [19, 51, 54] and/or APIs [2, 7, 9, 62, 64, 65]. However, due to the evolution of attack strategies of Android malware [55] classifiers built on these features may not be *sustainable*—they would need to be retrained constantly for later use, or their performance could downgrade over time.

The sustainability challenge with ML-based malware detection has been approached recently (albeit implicitly), yet with only limited investigation depth and solutions. For instance, *MamaDroid* [42] aims at sustainable detection by using features based on Markov modeling of abstracted API calls. While achieving high (up to 99%) accuracy when trained and tested on samples of same years, *MamaDroid* kept good performance for only one year. When trained on samples from year N, its detection accuracy dropped noticeably over time, to 75% for testing samples from year N+2 and below 50% for those from years N+3 and later. As another example, *RevealDroid* [29] attained 98% accuracy in a time-agnostic setting (i.e., the age of apps used in training and testing was dismissed), and the performance experienced significant drop to 87% in a time-aware setting (i.e., older apps were used for training and newer ones for testing with respect to a split date) within a short, three-year span.

In this paper, we extensively assess the sustainability of ML-based malware classifiers, in order to understand how to design more sustainable approaches to malware classification. First, motivated by the performance deterioration suffered both by existing malware detectors that motivated *MamaDroid* and by *MamaDroid* itself, we have performed a large-scale study of five state-of-the-art malware classifiers for Android, focusing on *how well* they may sustain their detection capabilities for a relatively long period of time. We found that all the classifiers deteriorated significantly (albeit not continuously) over years, and inspected *why* the deterioration occurred.

Then, motivated by our study findings, we propose a new behavioral profile for Android apps, called *sensitive access distribution (SAD)*, that models their run-time behaviors by capturing quickly-exposable patterns of access to sensitive data and operations through short tracing. Given an app, the SAD profile characterizes its execution in terms of the extent and distribution of invocations of sensitive data sources and sinks, and those of sensitive control flows at method level. Each SAD profile is described by a small set of 52 dynamic features, computed solely from the trace of *all* method calls in the app that are exercised during the tracing. Thus, the profile construction does not need static analysis but only straightforward, lightweight bytecode instrumentation. With respect to this dynamic profile, we have performed a *longitudinal* characterization of 13,627 benign and 12,755 malicious apps from across the past eight years. Based on the SAD profile, we developed *DroidSpan*, a novel system for malware detection aiming at *superior sustainability*.

Our characterization reveals that sensitive access was prevalently *executed* in all apps over all the examined years, with lower extent and frequency of use in benign apps than in malware, as expected. Also, in terms of the semantics of the access, network information and account settings were the top dominating data and operations involved, respectively, in both benign and malicious apps (with lesser extent of use in benign apps). Over time, benign apps tended to be much more stable than malware in terms of the SAD profile as a whole. Most notably, despite the evolution of Android and its apps, the average SAD profile of malware and that of benign apps were consistently *separable*, albeit the size of the differences fluctuated over the years.

We defined the *sustainability* property of app classifiers and proposed two metrics to quantify it. We then extensively evaluated the sustainability of *DroidSpan* against the same longitudinal datasets, with cross-validation and independent testing (i.e., predicting new/unseen samples). Our evaluation shows that *DroidSpan* achieves F1 accuracy superior to (by 6–32% on average) one state-of-the-art dynamic app classifier [3] and four state-of-the-art static approaches [9, 29, 42, 54]

for *same-period* detection (testing apps of the same year as the training data). For classifying apps appeared one to seven years later after training (*over-time* detection), *DroidSpan* also significantly outperformed all the five baseline techniques (by 21–37% on average) over all possible spans with respect to our datasets.

Significance. We note that sustainability of malware detection matters for their practical adoption. With new kinds of malware constantly emerging, an effective malware detector has to adapt to the new malware population in order to identify them on time. With low sustainability, a malware detector would typically rely on constant retraining using samples of newer apps. However, retraining may not be an option for two main reasons. First and foremost, in practice, samples of newer apps (new malware in particular, including zero-day malware) are often unavailable for retraining a detector such that the detector could recognize them. Second, retraining a learning model, including feature computation on new samples, can be highly expensive, especially when a large training set is required and/or a large number of features are used. Note that online/incremental learning [43] also is subject to the availability of new samples.

Rationale of Improvement. With a relatively intuitive and small set of features, *DroidSpan* has achieved superior classification performance in both same-period and over-time settings. This superiority is grounded in the fact that (1) the features underlying *DroidSpan* were discovered through an earlier study that showed they were able to significantly differentiate malware from benign apps, and (2) our longitudinal characterization with respect to these features as presented in this paper confirmed that they can consistently separate malware and benign apps when both groups evolve.

Contributions. This work makes the following contributions:

- First, we formulated the sustainability problem in malware detection techniques via ML-based classification, and defined two metrics (reusability and stability) to quantify the classification sustainability.
- Second, we conducted an extensive comparative study on sustainability of five state-of-the-art static and dynamic malware detectors for Android, and empirically revealed sustainability insufficiencies in existing approaches.
- Third, we developed a novel malware detection approach *DroidSpan* based on a new behavioral profile for Android apps, *sensitive access distribution*, defined by a very small feature set, which captures quickly-exposable sensitive access patterns in shortly traced app executions.
- Fourth, we evaluated the sustainability of *DroidSpan* against the five state-of-the-art approaches and showed its *statistically significant and large* improvements in that regard over those approaches with relatively low costs.
- Finally, we share open-source, full implementations of *DroidSpan* and the five existing detectors (i.e., the parts we added for the experiments in this paper) and all datasets, which facilitate replication of this work and development of future works on Android security. The entire artifact package can be found at https://bitbucket.org/haipeng_cai/droidspan.

2 SUSTAINABILITY OF MALWARE DETECTION

For ML-based app classifiers, we ask the question: *how long and well can a classifier sustain its performance with and without retraining on newer samples?* Accordingly, we define the sustainability in terms of two metrics:

- *Reusability.* The reusability of a classifier *cl* indicates how well *cl* can adapt to a new population *with* retraining on new samples from the population, and is measured by the classification accuracy e^r of *cl* in an average case when trained and tested on apps of the same year.

- *Stability.* The stability of a classifier *cl* indicates how stable *cl* is *without* retraining or any other model updates, and is measured by a tuple $\langle e^s, n \rangle$, where e^s is classification accuracy *cl* achieves in an average case when trained on apps of year x and tested on apps of year $x + n$, $n \geq 1$.

Intuitively, a key to achieving high sustainability of an ML-based app classifier lies in the underlying features (1) being able to differentiate benign apps from malware, and (2) capturing the change pattern of those features during the evolution of apps. Accordingly, our approach (1) incorporates various ways to model sensitive data access, and (2) uses relative statistics (e.g., ratio) instead of sheer numbers (e.g., number of API calls) to characterize app behavioral patterns.

In the rest of the paper, we first describe the design of *DroidSpan* in detail (Section 3), and then present the characterization of our study dataset in terms of the evolution-resilient features we discovered that consistently differentiate benign and malicious apps (Section 4). It is these features that enabled the improved sustainability of *DroidSpan*, which in essence explains why our detection approach worked better than the five baseline approaches we compared against it. Next, we present our extensive comparative study on the sustainability of the six detectors considered (Section 5). Although we actually performed the characterization study and the comparative study for the five baselines before developing *DroidSpan*, we describe the approach first for a more concise presentation structure. We further discuss why the baseline approaches had lower sustainability compared to *DroidSpan* (Section 6), followed by validity threats of our results as well as performance factors and limitations of *DroidSpan* (Section 8), before discussing related works (Section 9) and making concluding remarks (Section 10).

3 APPROACH

This section presents *DroidSpan*, our novel Android app classification system using supervised learning based on the defining features of our proposed SAD profile. The SAD profile is a new behavioral model of Android apps that characterizes their run-time access to sensitive data (by invoking sources) and operations (by invoking sinks). Next, we describe the defining metrics (i.e., features) of a SAD profile and how the profile is constructed for a given Android app. Finally, we describe the ML component of *DroidSpan*.

3.1 SAD Profile Definition

The SAD profile of an app is defined by 52 features, much simpler than the app profiles used by most existing ML-based app classifiers (which often use tens of thousands of features, e.g., 150K in the best-performing mode of *MamaDroid* and over 500K in *RevealDroid* for a dataset of 1,000 samples). The central focus of all these features is on the distribution of source/sink invocations, albeit they fall in three classes each addressing a different perspective: (1) the overall *extent* of sensitive access in terms of total source/sink invocations, (2) the *categorization* of invoked sensitive access in terms of the categories of data retrieved by the calls to sources, and the categories of operations performed by the calls to sinks, and (3) the method-level control *flows* that potentially carry out execution paths reaching to sink calls from source calls.

Table 1 gives a summary definition of the features that constitute a SAD profile, with the sizes (in parentheses) of each feature subset. All features are consistently percentages of calls in one group over those in total. We use these *relative* statistics instead of sheer numbers in order to capture the *general patterns* of app behaviors regarding sensitive access. We examine each app trace in two complementary views: *callsite* view concerning unique callsites regardless of the times a callsite is invoked, and *instance* view counting all call instances. For each feature in the callsite view (e.g.,

Table 1. Features (52) that constitue a SAD profile

Features	Description (all starting with "percentage of")
Extent of sensitive access	
Total source/sink callsites (2)	callsites targeting sources (resp. sinks) over all callsites.
Total source/sink call instances (2)	instances of calls to sources (resp. sinks) over all call instances.
Categorization of sensitive data and operations accessed	
Sensitive callsite distribution (11)	callsites targeting sources (resp. sinks) that are in each category (out of 5, resp. 6) over all source (resp. sink) callsites.
Distribution of sensitive call instances (11)	instances of calls to sources (resp. sinks) that are in each category (out of 5, resp. 6) over all source (resp. sink) call instances.
Sensitive (potentially vulnerable) method-level control flows	
Vulnerable source/sink callsites (2)	callsites targeting vulnerable sources (resp. vulnerable sinks) over source (resp. sink) callsites.
Vulnerable source/sink call instances (2)	instances of calls to vulnerable sources (resp. vulnerable sinks) over all source (resp. sink) call instances.
Vulnerable callsite distribution (11)	callsites targeting vulnerable sources (resp. vulnerable sinks) that are in each category (out of 5, resp. 6) over all vulnerable source (resp. sink) callsites.
Distribution of vulnerable call instances (11)	instances of calls to vulnerable sources (resp. vulnerable sinks) that are in each category (out of 5, resp. 6) over all source (resp. sink) call instances.

percentage of exercised callsites targeting a source), there is a *dual* feature in the instance view (e.g., *percentage of total instances of method calls that target a source*).

An app may invoke sources and sinks for legitimate purposes. Thus, source/sink invocations can only be regarded as *sensitive* but not necessarily *vulnerable*. We consider a source *vulnerable* if it reaches at least one sink callsite, and a sink *vulnerable* if there is at least one invoked source that reaches the sink, both through method-level control flows (i.e., dynamic call paths). While an imprecise approximation, the reachability based on control flows at method level is cheaper to compute and potentially more resilient to malware evasions [41], compared to tracking sensitive data flows [9]. Note that we do not count the numbers of sensitive flows (nor compute all of them) but only use the reachability in the SAD profile.

We categorize sensitive data access with respect to five categories of information that sources mostly retrieve (i.e., *Account*, *Calendar*, *Location*, *Network info*, and *System configurations*). We also categorize sensitive operations with respect to six categories of operations that sinks mostly perform (i.e., *Account setting*, *File operation*, *Logging*, *Network access*, *Messaging*, and *System setting*). We chose these categories because they were the predominant ones according to our previous empirical study in this regard [18].

The features defining SAD were originally discovered from an exploratory dynamic characterization study of a set of Android apps across years (entirely different from the longitudinal datasets used in this work). In that exploratory study, we experimented with a much larger set of candidate features [15] before finding the SAD features.

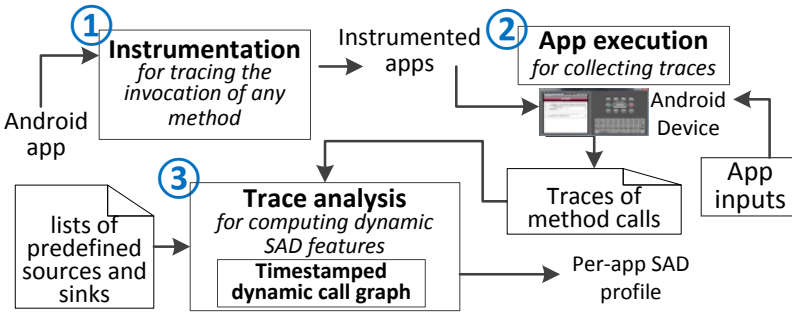

Fig. 1. The process flow of SAD profile construction.

3.2 SAD Profile Construction

Constructing the SAD profile for an app is reduced to extracting the 52 features from the app. Figure 1 depicts the construction process flow, with the following three major steps indicated by corresponding number labels. First, the app is instrumented for tracing. To overcome the vulnerability of static analysis to code obfuscation, we probe for invocation of all methods in the APK bytecode, including those in exception-handling constructs (e.g., `catch` and `finally` blocks) and those invoked via reflection.[1] Second, the instrumented app is exercised with automatically generated test inputs for T minutes on a device. Lastly, all the SAD features are computed from the collected trace. Currently, *DroidSpan* uses $T = 5$ to balance between overall cost and trace coverage—we tried shorter and longer tracing time lengths and found that 5 minutes represent a good balance. To identify the calls to sources and sinks from the trace, we use the source/sink lists generated by SUSI [47] and our manual categorization that refines the original one [15]. We use these conservative (thus relatively large) lists of predefined (18K) sources and (7.2K) sinks to consider access in apps sensitive very conservatively, which is necessary for reducing false negatives in our malware detection based on the use of sources and sinks.

From each app trace, a *timestamped dynamic call graph* (TDCG) is built to facilitate computing the features, especially those based on the statistics about *vulnerable* sources and sinks. A TDCG is a dynamic call graph with call-frequency and timestamp annotations. The graph collapses all instances of a method call, with a timestamp to denote the timing of each instance of the call. This design decision trades precision for efficiency (relative to an instance-level dynamic call graph). We use a straightforward timestamping algorithm that facilitates reachability computation: a trace-wise global counter is maintained, and for a call instance, the timestamp for the caller and callee is assigned c+1 and c+2, respectively, where c is the current value of the global counter. Thus, each node of the graph represents a callsite invoked as found in the trace. The size of the timestamp vector associated with a node indicates the call frequency of the callsite represented by the node.

All features concerning *sensitive* sources/sinks (the first 26 described in Table 1) are computed simply by counting the number of distinct sources/sinks and their total occurrences. For the reachability computation for the other 26 features concerning *vulnerable* sources/sinks, we first locate the lowest common ancestors of each pair of source and sink on the TDCG. Then, we compute the control flow path from the source to the sink, through backward traversal from the sink to the ancestor and forward traversal from the ancestor to the source, using the timestamps associated with each callsite (i.e., a call with a smaller timestamp is not reachable from a call of a greater timestamp). To trade completeness for efficiency, the analysis underlying the TDCG construction is

[1]That is, we do not recognize source/sink APIs in the source code during instrumentation; instead, all callsites are probed for monitoring the call targets, and source/sink calls are recognized later only from the call traces.

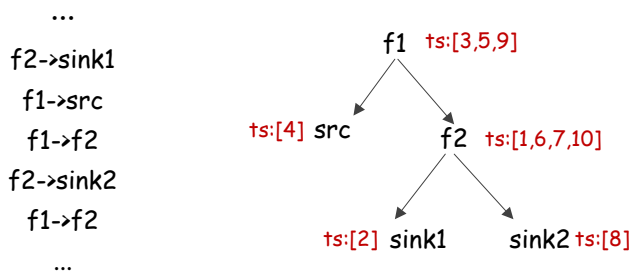

Fig. 2. A *DroidSpan* trace excerpt (left) and the corresponding timestamped dynamic call graph (right).

purely dynamic, capturing only the control flows due to ordinary function calls. As a result, control flows induced through callbacks, inter-component communication (ICC) [35], multi-threading[2], and message passing are dismissed—capturing those would require non-trivial static analyses hence compromise the efficiency and scalability of *DroidSpan*.

Illustration. Figure 2 shows an illustrative trace (left) and the TDCG (right) built from the trace. Each line of an app trace simply records a call instance in the format of *caller*→*callee*. In the TDCG, each timestamp vector (*ts*) annotating a node maintains the timestamp for each instance of the node. $f1$ and $f2$ are two user-defined methods, *src* is a source that retrieves *Account* information, and *sink*1 and *sink*2 are two sinks that perform *Logging* and *Messaging*, respectively. Since there are 1 source and 2 sink callsites, one call instance of each source/sink callsite, 5 total callsites (each corresponding to a node on the right of the figure), and 10 total call instances (i.e., some of the callsites are invoked multiple times; e.g., the callsite in $f1$ that targets $f2$ are invoked four times, with timestamps 1, 6, 7, and 10), the features for the *extent of sensitive access* are (1/5, 2/5, 1/10, 2/10). Given the categories of these sources/sinks and the total of five source and six sink categories we considered (Section 3.1), the sensitive callsite distribution features are (1/1, 0, 0, 0, 0) for sources and (0, 0, 1/2, 0, 1/2, 0) for sinks. Since each source/sink is executed once here, the 11 features for distribution of sensitive call instances are the same. In this example, *sink*2 is reachable from *src* (e.g., via a path "*src* [ts=4] → $f1$ [ts=5] → $f2$ [ts=6] → *sink*2 [ts=8]"), yet *sink*1 is not (with respect to the timestamps: *sink*1 was never invoked later than any instance of *src*). Thus, there is only one callsite (and instance) of source (and sink) that is *vulnerable*, hence the features on vulnerable source/sink callsites and instances are (1/5, 1/5, 1/10, 1/10). The vulnerable access distribution features are (1/1, 0, 0, 0) for sources and (0, 0, 1/2, 0, 0, 0) for sinks, same as the features on the distribution of vulnerable call instances.

3.3 SAD-based Classification

DroidSpan is a dynamic malware detector based on the SAD profile. Like existing ML-based app classification systems, *DroidSpan* works in two major phases: training and testing. In the training phase, it relies on a set of labeled app samples to build a prediction model. Then in the testing phase, the trained model is applied to classify novel apps. Our main goal with and motivation for *DroidSpan*, however, is that this model can be used effectively (i.e., with improved classification performance) some time (years) after it is trained, so that *emerging* malware can be discovered by the system without being retrained on samples for the new malware. Intuitively, the longer the time span, the better, hence the more sustainable the system.

[2]For instance, a sink exercised in one thread would not be considered vulnerable if it is only reachable from a source exercised in a different thread.

To make the prediction model sustainable, the key is that the underlying features can, for a relatively long period of time, distinguish benign apps from malware. To that end, *DroidSpan* uses the features that define the SAD profile of an Android app (Table 1). Our longitudinal characterization (Section 4.2) shows that, with the evolution of malware attack strategies and the Android system, the behaviors of benign apps and malware developed in varying years were consistently *separable* in terms of their SAD profiles, despite fluctuations in both groups over time. These findings provide the fundamental grounds for *DroidSpan* to achieve improved classification sustainability over years.

Feature extraction. Extracting the SAD features is needed both for obtaining the sample set for training *DroidSpan* and for later predicting the label of a novel app using *DroidSpan*. We compute the features of all training and testing apps as described earlier (Section 3.2).

Classification. To find the best ML model for *DroidSpan*, we have performed a comparative study of *DroidSpan* variants using most of the supervised classification models provided in the Scikit-learn library [45]. These models include: random forest (RF), support vector machines (SVM) with both linear and radial basis function kernels, decision trees/C4.5, k-nearest neighbours (k-NN), naive Bayes with three possible probability models (Gaussian, Multinomial, and Bernoulli), AdaBoost, Gradient Tree Boosting, Extra Trees, and the Bagging classifier. By comparing these variants of *DroidSpan* with respect to classification performance (precision, recall, and F1-measure), we found that RF consistently achieved the best performance on any of our datasets. Thus, we chose RF as the learning algorithm for *DroidSpan*.[3] *DroidSpan* is first trained on benign apps labeled as BENIGN and malicious apps all labeled as MALICIOUS (despite the varying families the malware belongs to), and then classifies a novel app as either of the two labels.

Applicability Scope. The classification capability of *DroidSpan* is based on its modeling of app behaviors that are (implicitly) relevant to maliciousness in terms of sensitive data access. Thus, it might seem that this SAD-based behavior modeling only addresses sensitive information leaking/stealing. We note that an SAD profile essentially characterizes information flow from certain sources to sinks in general, instead of only concerning information leaking/stealing. Information flow security is a fundamental aspect of software security [50] that deals with a variety of specific security threats (e.g., known as taint-style security issues, including SMS trojans and ransomware attacks [6]). Typically, a specific information flow security problem can be tackled by looking at particular kinds of information flow of interest (e.g., file-encrypting flow in ransomware and flow paths leading to SMS invocations in SMS trojans). Therefore, an SAD-based classification like *DroidSpan* is more broadly applicable to malware of varies kinds than information leaking/stealing malware. In fact, as shown in Table 2, our malware datasets represent a large variety of malware in terms of malware families, thus the promising performance results of *DroidSpan* suggest the wide applicability of the SAD-based classification.

4 STUDY DATASETS & CHARACTERIZATION

To understand how sensitive access in Android apps evolves, we collected real-world app samples from early 2010 to late 2017, and characterized them in terms of their SAD profiles.

4.1 Datasets

Table 2 gives an overview of our study datasets, including total numbers of samples downloaded (*#samples*), and numbers of samples that were actually used (*#used*). Particularly for malware, the

[3]This model selection strategy has been commonly used in peer work as well—for instance, *MamaDroid* [42] chose RF also after considering 1-NN, 3-NN, and SVM; *DroidSieve* [54] chose Extra Trees after trying with RF, SVM, and eXtreme Gradient Boost; *Afonso* [3] chose RF after comparing with J.48, NaiveBayes, SimpleLogic, and a few others.

Table 2. Overview of our study datasets

Benign apps				Malware				
Name	**Year**	**#samples**	**#used**	**Name**	**Year**	**#samples**	**#used**	**#families**
B10	2010	1,530	1,344	M10	2010	2,029	1,877	78
B11	2011	2,019	1,757	M11	2011	1,431	1,303	116
B12	2012	2,053	1,845	M12	2012	2,225	1,945	207
B13	2013	1,748	1,568	M13	2013	1,230	1,139	208
B14	2014	3,127	2,953	M14	2014	1,493	1,337	181
B15	2015	1,333	1,178	M15	2015	1,667	1,451	322
B16	2016	1,548	1,370	M16	2016	2,171	1,769	224
B17	2017	1,650	1,612	M17	2017	2,205	1,934	263
total		*15,008*	*13,627*	*total*		*14,451*	*12,755*	*917*

last column (*#families*) shows the number of malware families represented by the malicious samples used per year. In this paper, we focus on malware detection only and do not address malware family classification. Nevertheless, we wanted to ensure the diversity of the malware samples included so that we may validate that the presented methodology and technique are not only applicable to a few specific kinds of malware.

For each yearly dataset (named in the first and fifth columns), the samples on which at least one of the six techniques (*DroidSpan* and five baselines) did not successfully compute features were removed from the downloaded set: some of the apps cannot be unpacked, miss asset files, or cannot be processed by one or more of the compared techniques. Of these 16 datasets, *B17* was obtained by downloading the top 100 popular apps in each of the app categories on Google Play. *M13, M14, M15* and *B16* were obtained from VirusShare [34]. All the other datasets were obtained from the AndroZoo suite [4]. We considered different data sources to make our benchmarks more diverse hence more representative. Our entire datasets include 15,008 benign and 14,451 malicious samples, for a total of 29,459 benchmarks. Out of these apps, 26,382 (13,627 benign apps and 12,755 malware) were used. The 12,755 malicious samples represent 917 unique malware families. All the 16 datasets are mutually disjoint (there were no apps shared by any two datasets).

We produced the SAD profile of each used sample by running the instrumented app on an Android emulator (Nexus One), with SDK 6.0 (API 23), 2G RAM, and 1G SD storage. The emulator ran on an Ubuntu (15.04) desktop of 8G memory and 8-core 2.6GHz CPU. Each app was exercised for five minutes by the random inputs generated by Monkey [30]. We used Monkey because it achieves an average code coverage at least comparable to other existing alternatives (research prototypes) while with the best usability and reliability [20, 44]. To minimize bias, we run each app in a fresh emulator setting (by restarting a clean emulator for each app).[4] The line coverage of the per-sample traces averaged 48.2%, with standard deviation of 22.1% and median of 50.1%. With our toolkit, a real Android device can be easily plugged in to produce traces for *DroidSpan*. Since it is known that malware may hide their behaviors when they detect being run on an emulator [48], we purposely chose an emulator over a real device in our experiments so that we can check by the way the robustness of our dynamic detection approach against those evasion schemes.

[4]Note that keeping a consistent execution environment (e.g., emulator settings) is essential for the stability of *DroidSpan*'s computation of app feature values—as an counterexample, in emulators with contact lists of quite different lengths, for many of its features, *DroidSpan* could compute very different values across the emulators because the app exerciser (either an automated input generator or a human) would trigger the invocation of relevant callsites for largely varied numbers.

4.2 Datasets Characteristics: SAD Evolution

As mentioned earlier, in order to discover the evolution-resilient discriminatory features for *DroidSpan*, we conducted an exploratory dynamic characterization study [15] with respect to a large set of metrics. To avoid potential biases in our evaluation (Section 5) of the chosen baseline malware detectors against *DroidSpan*, we used a smaller datasets that are disjoint from any of the datasets of Table 2. We selected 52 out of those metrics for which the mean metric value of the benign-app group was consistently greater or smaller than that of the malware group for any year, and used the 52 resulting metrics as the features to form the SAD profile.

Next, we present and discuss the evidences of these SAD features being able to consistently distinguish malware from benign apps, according to the two classes these features fall into: *source/sink use extent* and *used source/sink categories*. For illustration purposes, we chose two SAD features as examples to depict how the SAD profile was resilient against the evolution of both benign-app and malware groups over the eight years we considered. This characterization on SAD evolution essentially enabled us to understand the evolutionary patterns of sensitive accesses in Android apps.

Evolution in source/sink use extent. We examined the evolutionary patterns of the extent of sensitive accesses in terms of source and sink calls in both the *callsite* and *instance* views. We found that the sensitive accesses were substantial (given our conservative definition of such accesses) in all the benchmarks studied, benign or malicious. On the other hand, benign apps generally tended to exercise more distinct sources and sinks (evidenced by higher percentage of source/sink callsites out of all callsites exercised), while malware tended to invoke sources/sinks more frequently (indicated by higher percentages of source/sink call instances out of all instances of method calls). A plausible explanation for this contrast is that many malicious behaviors in mobile apps are related to accessing sensitive data (via calling sources) and/or operations (via calling sinks), as we found earlier [14, 23]. Importantly, although the average feature values fluctuated noticeably across years in both the benign-app and malware groups with respect to each of the 8 features on source/sink use extent, the signs of the differences between the two groups were *consistent over the years*: for each feature, the value of benign apps was consistently higher/lower than that of the malware in any of these years. This consistent separation between malware and benign apps by a feature allows for a classifier that uses the feature to differentiate the two groups against evolution.

Figure 3 illustrates these observations with respect to two SAD features: *the percentage of sink callsites* (top) and *the percentage of vulnerable call instances* (bottom). In each chart, we show the average-case SAD feature values for benign versus malicious apps. Since we were interested in comparing the means, we computed the 0.95 confidence interval for each mean (represented by the error bars) to estimate how well these sample means approximate those of the respective populations. We used the Vysochanskij-Petunin inequality [59] to compute the non-parametric intervals since we cannot make assumptions about the normality of the data distribution. Considering how the intervals are computed using the inequality, for a given set of samples, the width of such an interval of the sample mean immediately reflects the sample variance. As shown, while benign apps had more distinct callsites targeting a sink, malware had higher frequency of calls to sinks that are vulnerable, as mentioned earlier. This implies that malware tended to focus on a smaller set of sinks with each intensively invoked, when compared to benign apps.

Notably, there was no intersection between the trend curves of the two app groups. In an average-case the feature value of benign apps was always greater (for *the percentage of sink callsites*) or smaller (for *the percentage of vulnerable call instances*) than that of malware, although the difference sizes varied and were small at some years.

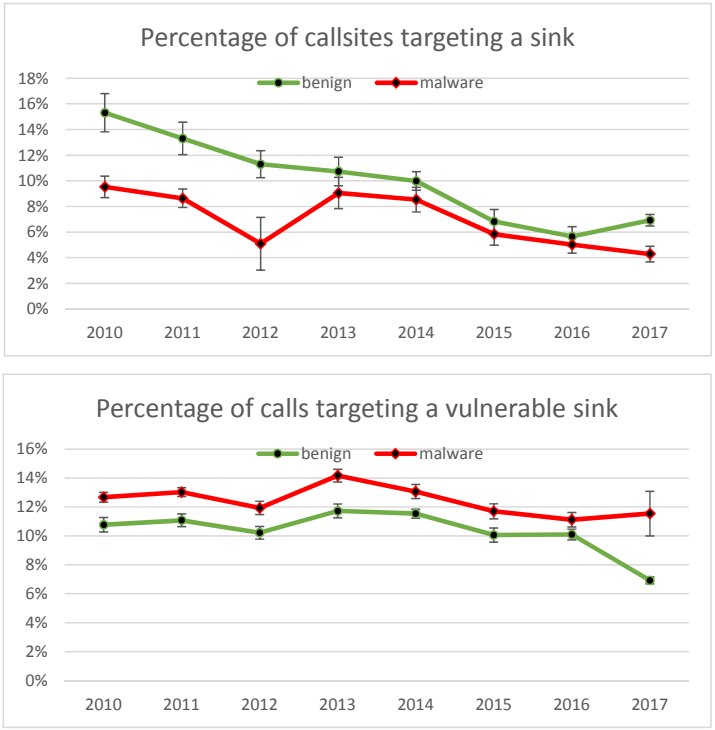

Fig. 3. Part of our SAD characterization results: mean percentage of calls for sensitive access (y axis) over years (x axis) in benign apps (blue) and malware (orange). Error bars show 0.95 confidence intervals of the means (with the interval width immediately reflecting the sample variance).

Evolution in used source/sink categories. Most of the features in our SAD profile concern the categories of source/sink APIs associated with the sensitive access. Our results clearly indicate that in both malware and benign apps, among all sources exercised, the dominating ones were those retrieving network information, while calls to sinks related to account settings[5] dominated all sink calls. This observation was consistent regardless of the views and whether the sources/sinks were vulnerable or not. More importantly, like features on source/sink use extent, these categorization based features were consistently able to distinguish between the benign and malicious apps over the eight years, albeit the gaps were larger in some years than in others.

5 EXPERIMENTS AND RESULTS

This section presents our experimental methodology and results of evaluating the sustainability of state-of-the-art app classification approaches in contrast to *DroidSpan*.

5.1 Study Setup

We assessed *DroidSpan* in terms of classification performance and efficiency, versus *MamaDroid*, which is a state-of-the-art ML-based malware detection system and shares a similar goal to ours

[5]Our source/sink categorization originated from SUSI [47] which assigns categories for source/sink APIs in a conservative and approximate manner.

(i.e., sustainable detection[6]). Also, both systems use features based on API calls in an app, albeit *MamaDroid* extracts features with static analysis, using FlowDroid [8] to build call graphs while *DroidSpan* is based on pure dynamic analysis. Both systems use RF as the learning algorithm for building the app classifier.

We have not been aware of other prior works that explicitly reported sustainability results, yet it is possible that some of them may have good sustainability. Thus, we included four more baseline techniques: (1) *DroidSieve* [54], another state-of-the-art static approach that uses features computed from app resources (e.g., asset files) and Extra Trees for classification, (2) *Mudflow* [9], a one-class SVM classifier that builds the classification model based on features of benign samples only that characterize data flows from sources to sinks (computed by FlowDroid), (3) *Afonso* [3], a state-of-the-art dynamic approach that classifies apps based on calls to predefined lists of APIs and system calls using RF, and (4) *RevealDroid* [29], a lately proposed approach to app classification based on apps' usage of APIs, native code, and reflection, also using SVM for classification. For *Afonso*, we implemented the entire tool according to the lists of system calls and Android SDK APIs monitored by the approach as given in the paper [3]. We had to reimplement the tool because we were not able to access it from a publicly available source. For *Mudflow*, we were able to reuse the entire tool except for a few scripts for computing the taint flow paths using FlowDroid and formatting the feature data to fit the tool. For other three techniques, we obtained the code for feature computation from the authors (i.e., the core part of the tools) and used the code exactly; we had to only implement the ML parts because they were not available for reuse— The original ML part of *RevealDroid* was available but partially hard-coded for original experiments, which we cannot immediately use.

We computed the features of these baselines using corresponding tools with the exact configurations described in the respective original papers (e.g., tracing each app on an Android emulator against Monkey inputs for 5 minutes for *Afonso*). With help of their authors, we were able to replicate the performance results against part of the datasets originally used, hence gained confidence about the correctness of our tool setups. Given the large number of malware detectors in the literature, we limited our baselines to the ones that appeared more lately and surpassed many prior ones (e.g., *RevealDroid* over *Drebin* [7]).

We conducted two studies. In *Study I*, we assess the performance of *DroidSpan* with training and testing apps developed in a same period of time (referred to as *same-period detection*). The goal of this study is to assess the *reusability* of app classifiers. In *Study II*, we focus on evaluating the *stability* of the six tools by assessing their performance when they are trained on older datasets and predict labels of newer ones, spanning one to seven years (referred to as *over-time detection*). We used the benchmarks listed in Table 2 for both studies. *MamaDroid*, *Mudflow*, and *RevealDroid* require large amount of memory, due to which we computed features and classification results on a HP DL580-G7 server with 256GB memory.

5.2 Methodology

We gauged the performance of *DroidSpan* versus the baselines in terms of precision, recall, and F1-measure (accuracy). Give an app p under classification, p is a *true positive* (TP) if the true label is L and the predicted class label is L; it is a *false positive* (FP) if the true label is not L and the predicted label is L; it is a *true negative* (TN) if the true label is not L but the predicted label is not L; and it is a *false negative* (FN) if the true label is L and the predicated label is not L. Accordingly, regardless of the number of classes (distinct labels) in the model: Precision (P) = $\frac{TP}{TP+FP}$, Recall (R) = $\frac{TP}{TP+FN}$, and F1

[6]Sustainability was not explicitly addressed nor quantified originally for *MamaDroid* [42] but its motivation was similar in spirit to *sustainable* malware detection as we focus on in this work.

$= 2 * \frac{P*R}{P+R}$. For each testing, we computed the weighted average of each of the three metrics over the two classes in malware detection (i.e., BENIGN and MALICIOUS), where the weights are class sizes. We used these average metrics (over both malware and benign classes) in order to measure not only malware detection rate (i.e., true positive rate, which concerns the malware class only as used as the primary evaluation metric in prior work on malware detection [7, 9, 19, 54, 63]), but also the ratio of false alarms (concerning the benign class) which immediately affects user-inspection effort—inspecting benign apps that were falsely flagged as malware is wasteful.

In *Study I*, we conducted 8 rounds of training and testing, each using the benign-app and malware datasets from the same year (e.g., B10+M10, meaning benign apps of year 2010 combined with malware of year 2010). Specifically, we randomly selected a third of samples from each class (malware or benign) and reserved them as unseen/novel samples. By doing so, we intend to test the capability of a technique in classifying apps that it never used in training (albeit from the same time period as the training set). Concerning possible biases in the random selection, we also performed 10-fold cross-validations (CV) against the baselines as a smoothing scheme. For *Study II*, we conducted 28 rounds of training and testing, each using the benign-app and malware datasets of year x for training and those of year $y \in [x+1, 2017]$ for testing (e.g., training on B15+M15 and testing with B16+M16 and B17+M17, respectively). For fair comparisons, we produced the classification results of the six techniques from exactly the same apps for each training-testing round. In the case of a technique working at multiple modes/settings, we used the best results it produced on each dataset. We also performed two statistical analyses to assess the statistical significance and sizes of performance differences between *DroidSpan* and all the baselines.

5.3 Study I: Same-Period Detection (Reusability)

In this study, we assessed the reusability of the six compared techniques through evaluating their classification performance for same-period detection. A more reusable app classifier achieves higher F1 accuracy in an average case when it is applied to datasets from varying years.

Table 3. Reusability of *DroidSpan* versus the five baselines on datasets over eight years (hold-out validation).

Dataset	*DroidSpan*			*MamaDroid*			*DroidSieve*			*Afonso*			*RevealDroid*			*Mudflow*		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
B10+M10	0.9376	0.9360	**0.9362**	0.8424	0.8357	0.8367	0.8353	0.9347	0.8822	0.8788	0.8710	0.8718	0.8600	0.8540	0.8549	0.5246	0.5319	0.5065
B11+M11	0.9432	0.9417	0.9413	0.9893	0.9893	**0.9793**	0.9583	0.7091	0.8151	0.8978	0.8978	0.8978	0.8700	0.8641	0.8616	0.4598	0.4537	0.4563
B12+M12	0.9424	0.9424	**0.9423**	0.8378	0.8378	0.8377	0.9203	0.8000	0.8560	0.8954	0.8935	0.8935	0.8283	0.8279	0.8277	0.7344	0.6419	0.6450
B13+M13	0.9554	0.9529	**0.9525**	0.9141	0.9076	0.9060	0.9935	0.8102	0.8926	0.9217	0.9182	0.9172	0.8915	0.8823	0.8830	0.6362	0.6433	0.6311
B14+M14	0.9302	0.9272	**0.9250**	0.8462	0.8467	0.8449	0.8981	0.4528	0.6020	0.8673	0.8693	0.8665	0.8360	0.8389	0.8367	0.7040	0.7048	0.6930
B15+M15	0.9061	0.9042	**0.9036**	0.8450	0.8440	0.8442	0.8162	0.9193	0.8647	0.7798	0.7610	0.7514	0.8236	0.8014	0.7939	0.7213	0.7218	0.7125
B16+M16	0.9352	0.9342	**0.9339**	0.9021	0.8969	0.8955	0.8275	0.9787	0.8968	0.8138	0.8068	0.8025	0.8660	0.8444	0.8389	0.7532	0.5936	0.6135
B17+M17	0.9723	0.9720	**0.9720**	0.9126	0.9093	0.9098	0.8910	0.8892	0.8891	0.9510	0.9493	0.9493	0.8546	0.8360	0.8334	0.8331	0.7105	0.6668
Average	0.9408	0.9393	**0.9388**	0.8835	0.8810	0.8794	0.8929	0.7956	0.8271	0.8780	0.8738	0.8719	0.8523	0.8431	0.8408	0.6761	0.6284	0.6185

Table 3 lists the precision (P), recall (R), and F1 accuracy of each of the 8 independent tests (noted in first column) achieved by *DroidSpan* versus the five baseline techniques (first row). For instance, when using two thirds of the benign apps in B10 and two thirds of the malware in M10 for training, *DroidSpan* had an F1 accuracy of 93.62% in predicting the labels of one third of unknown samples of respective classes in the same datasets. This accuracy was higher than 83.67% by *MamaDroid*, 88.22% by *DroidSieve*, 87.18% by *Afonso*, 85.49% by *RevealDroid*, and 50.65% by *Mudflow*. Note that each F1 number was not computed from the associated precision and recall, but was the average of respective metric values for the two app classes (BENIGN and MALICIOUS). The last row shows the averages of classification performance results over the 8 tests, weighted by the number of testing samples in each test, which indicate the *reusability* of these compared techniques.

As shown, *DroidSpan* achieved precision, recall, and accuracy all ranging from 90% to 97% for the datasets of any of the past eight years. The classification performance had very small variations, indicating generally good reliability of *DroidSpan*. In comparison, the baselines all had relatively larger variations with any of the three metrics, indicating the greater dependence of their performance on particular datasets. On the other hand, *DroidSpan* outperformed all the baselines in all cases, with the only exception of *MamaDroid* on one single case of B11+M11, in terms of the F1 accuracy (in all cases, the best F1 is highlighted in boldface). *DroidSpan* performed the worst on B15+M15, which was also nearly the most-challenging datasets for most of the baseline techniques. As an overall average, the **reusability** of *DroidSpan* was 93.88%, versus 87.94% by *MamaDroid*. *Afonso* achieved slightly lower reusability than *MamaDroid* (87.19%), but higher than the other three baselines. *Mudflow* had the lowest performance among all the six detectors with a substantial gap in any of the three metrics. With respect to the originally reported performance, it was indeed the least accurate (around 86% versus above 95% in other baselines, albeit not based on the same datasets). The gap here in our study based on the same dataset suggests that modeling the behavior of only one (benign) class can face a great reusability challenge.

Table 4. Reusability of *DroidSpan* vs *MamaDroid* (10-fold CV)

Dataset	*DroidSpan*			*MamaDroid*		
	P	R	F1	P	R	F1
B10+M10	0.9251	0.9211	**0.9215**	0.8498	0.8400	0.8482
B11+M11	0.9305	0.9291	0.9286	0.9916	0.9908	**0.9901**
B12+M12	0.9312	0.9309	**0.9309**	0.8628	0.8630	0.8637
B13+M13	0.9459	0.9424	**0.9417**	0.9225	0.9156	0.9157
B14+M14	0.9113	0.9086	**0.9055**	0.8399	0.8381	0.8387
B15+M15	0.8960	0.8943	**0.8937**	0.8450	0.8412	0.8403
B16+M16	0.9258	0.9245	**0.9241**	0.8961	0.8845	0.8902
B17+M17	0.9632	0.9622	**0.9622**	0.9175	0.9128	0.9098
Average	0.9288	0.9268	**0.9261**	0.8881	0.8834	0.8848

As a smoothing strategy, we also performed a 10-fold CV per detector and yearly datasets. Table 4 lists the P, R, and F1 of *DroidSpan* against *MamaDroid*, the most competitive baseline in the hold-out validation. As shown, for both detectors the results were very similar between the two validation settings (see Table 3). While not shown, this is also the case for the other four baselines. The average, also weighted by the testing set sizes, shows similar reusability contrasts between *DroidSpan* and the baselines.

Conclusion. Our results revealed that *DroidSpan* achieved competitive reusability of 94% with small variations across years, outperforming all the five baselines considered in our study (by 6–32%). Thus, with newer samples available for retraining, it is promising to use *DroidSpan* continuously into the future.

5.4 Study II: Over-Time Detection (Stability)

This study aims at evaluating the *stability* of *DroidSpan* versus the baselines. We consider all 28 possible independent tests with our datasets that each trains a classifier on older benign and malicious apps while testing on at least one-year newer apps. We report P, R, and F1 for each test and compare them among the six detectors.

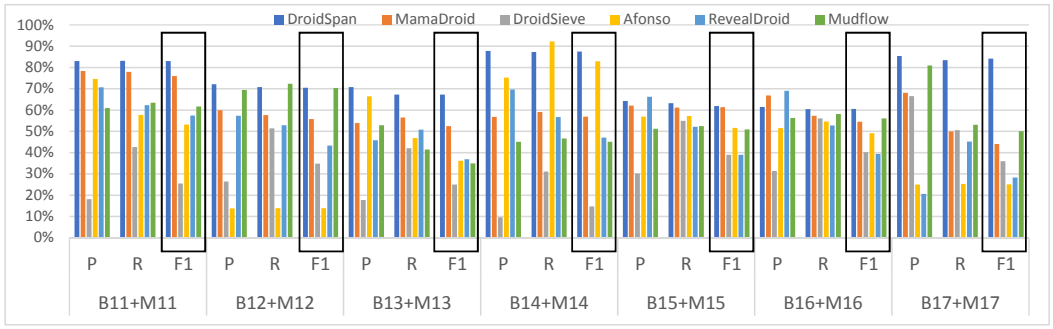

Fig. 4. Performance (y axis) of the six techniques compared, trained on B10+M10 and tested on apps of all possible next years with respect to our datasets (x axis).

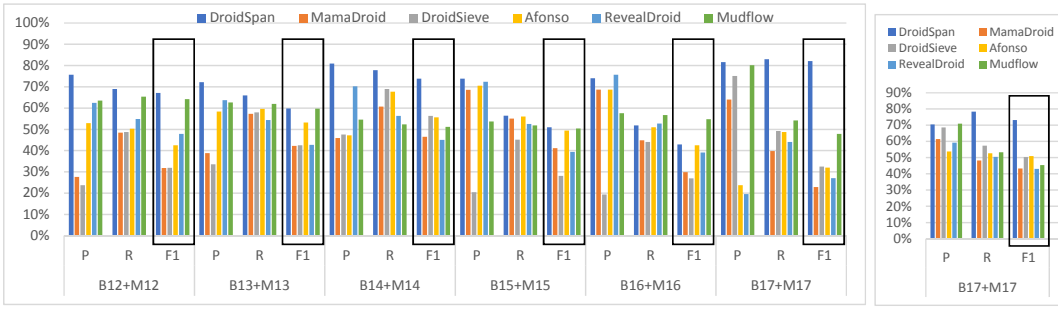

Fig. 5. Same format as Figure 4 but trained on B11+M11 (left) and B16+M16 (right).

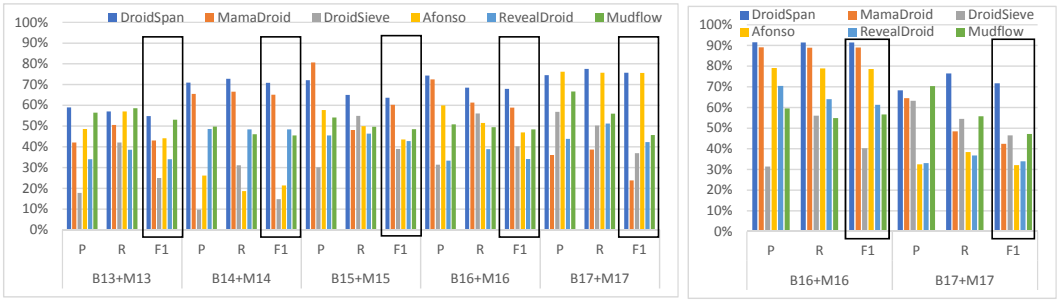

Fig. 6. Same format as Figure 4 but trained on B12+M12 (left) and B15+M15 (right).

Classification performance. Figures 4 through 7 show the performance metrics values resulted from all the 28 tests. Each figure compares the six techniques in terms of the three performance metrics, with one pair of (benign-app and malware) training datasets from one year (as noted in the figure caption) and the pairs of testing datasets from all possible later years (as noted under the x axis). For instance, Figure 4 depicts the performance results for the test in which B10+M10 was used for training the classifiers, while samples in each of the following seven years were used to test the trained detectors. To save space, each of the other three figures shows the comparative results from *two* tests. F1 bars are framed by rectangles to ease comparisons.

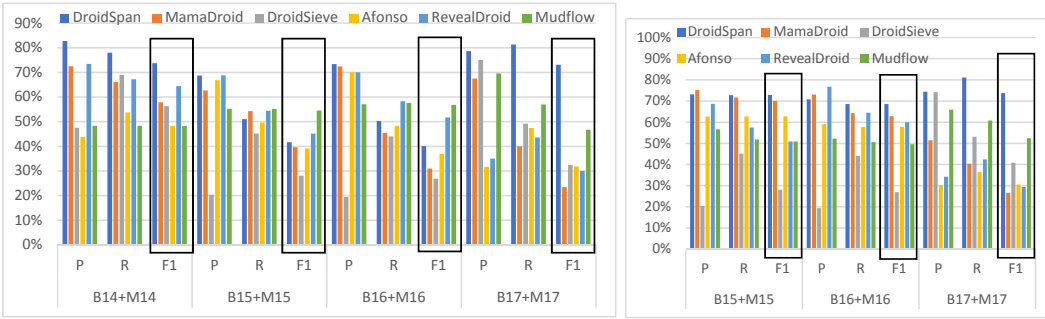

Fig. 7. Same format as Figure 4 but trained on B13+M13 (left) and B14+M14 (right).

Overall, the performance of any technique varied noticeably when being tested on datasets of different years after it was trained on the datasets of a specific year. For instance, after being trained on B10+M10, most of the techniques worked the best on apps from year 2014, as shown in Figure 4, while when trained on B11+M11 and B12+M12, the best performance of *DroidSpan* was seen by apps of year 2017, as shown in Figures 5 and 6, respectively. In terms of the performance numbers, these figures clearly show that over-time detection is much more challenging than same-period detection for all the techniques studied. For example, in most cases, even the best F1 accuracy out of the six techniques was below 80% in this setting, significantly lower than an average-case F1 accuracy seen in the same-period detection. On the other hand, while the performance of all the classifiers degraded from same-period to over-time settings, the degradation was not monotonic.

Importantly, *DroidSpan* had higher accuracy than all baseline techniques in 25 of the 28 tests, albeit the precision or recall was slightly lower than one or two baseline techniques in a small number of cases. Also, the improvements of *DroidSpan* over these peer techniques were mostly substantial, or at least noticeable. In the three exceptional cases (trained on B11+M11 and tested on B16+M16, trained on B13+M13 and tested on B15+M15 and B16+M16), *DroidSpan* was outperformed by *Mudflow* with moderate/appreciable differences (around 10% in F1). These three cases were also the worst cases for *DroidSpan*. An intuitive reason is that the SAD profile differences between the benign-app and malware groups of years 2015 and 2016 were generally much smaller than those in other years—for instance, with respect to the two SAD features we illustrated in Figure 3, the differences between the two groups were among the smallest. For apps of these years (and likely those of future years alike), according to our methodology for improving sustainability, further app evolution studies may be needed to discover more discriminatory features.

Among the five baseline techniques considered, *Mudflow* performed the best in most cases, followed by *MamaDroid*, *Afonso*, and then *RevealDroid*. As in the same-period detection setting, *DroidSieve* again had the lowest classification performance for over-time detection. *Mudflow* also had the smallest gap between these two detection testings among all these malware detectors compared, followed by *DroidSpan*. In fact, *Mudflow* appeared to be consistent in its classification performance across the two settings (in terms of the gap magnitude), yet unfortunately the performance was consistently low (mostly 50–60%). *DroidSieve* saw the largest performance gap between the two settings, indicating its greatest overfitting to the features of apps from same years.

Average Stability. To obtain an average-case view of the stability of *DroidSpan* versus the baseline techniques, we computed the mean F1 accuracy of each technique from all the 28 tests with one- through seven-year spans between the training and testing data. Figure 8 illustrates the average stability according to our over-time detection study for different spans (i.e., the numbers of years,

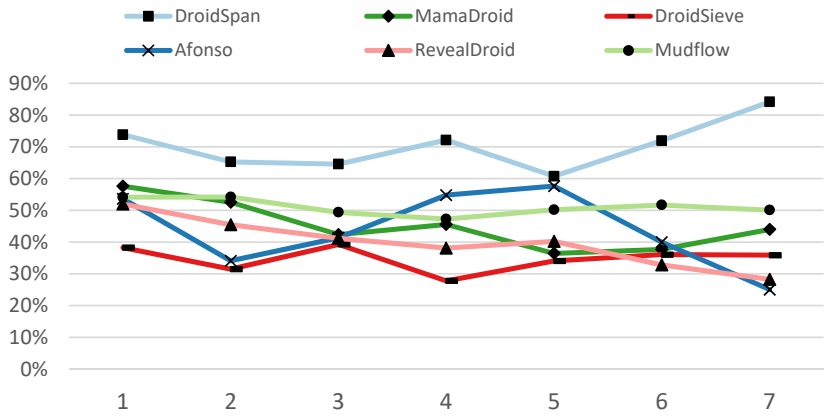

Fig. 8. Average stability with accuracy (y axis) and different spans (i.e., number of years after training, x axis) among the six techniques compared.

Table 5. Overall stability of *DroidSpan* versus the five baselines

DroidSpan	*MamaDroid*	*DroidSieve*	*Afonso*	*RevealDroid*	*Mudflow*
71.81%	42.43%	34.59%	41.72%	36.24%	50.39%

listed on x axis). As shown, *DroidSpan* clearly outperformed all the baseline techniques at any of the seven spans. For detecting malware appeared five years later after training, the accuracy improvement of our technique over the best performing baseline for that span (i.e., *Afonso*) was relatively small. At all other spans, however, the improvements of *DroidSpan* over any of the baseline techniques were much more substantial. As seen earlier, *Mudflow* performance was relatively consistent across these spans (i.e., for different span lengths), and it was the best-performing baseline, for the over-time detection. Recall that *Mudflow* had the lowest performance in the same-period detection setting. The contrast implies that, while learning only from the benign class may suffer from low classification accuracy, doing so benefits stability against app evolution. An intuitive justification is that since no malware is used for training, the classifier is not subject to overfitting with malware samples as are two-class classifiers. Among the remaining four baselines, there was no any technique that consistently outperformed the other three at all the spans. On the other hand, *DroidSieve* had the lowest accuracy at most (five) of the seven spans. Table 5 shows the average stability over all these spans. The overall average F1 of *DroidSpan* was 72%, followed by *Mudflow*, *MamaDroid*, *Afonso*, *RevealDroid*, and *DroidSieve*, in that order, consistent with the contrasts observed from Figure 4 through Figure 7.

Conclusion. When switched from same-period detection to over-time detection, all the six detectors degraded in performance substantially. Yet, degradation was neither monotonic nor linearly correlated with the length of span. On the other hand, among these compared detectors, *DroidSpan* noticeably outperformed all the five baselines in terms of stability at any span. Overall, *DroidSpan* improved over the baselines by 21% to 37% in terms of average stability.

5.5 Overall Sustainability

Putting results of Studies I and II together revealed that our technique consistently surpassed the four state-of-the-art learning-based app classifiers, static or dynamic. To further corroborate the superiority of our approach, we performed two statistical analyses. First, we performed a

set of paired Wilcoxon signed-rank tests [60] in which the two groups were the F1 scores given by *DroidSpan* and each baseline contrasted (α=.05). We adopted this nonparametric test to avoid assuming about the normal distribution of the F1 scores. The goal is to see whether the improvements of our approach are statistically significant. Second, we computed the Cliff's delta [21] (in a paired setting with α=.05), a nonparametric effect-size metric, to gauge the magnitude of those improvements our approach made over the baselines.

Table 6. Statistical differences in F1 between *DroidSpan* and baselines

Contrast Group	Reusability		Stability	
	p-value	Effect size	p-value	Effect size
DroidSpan vs *MamaDroid*	4.23E-02	0.75	4.00E-06	1
DroidSpan vs *DroidSieve*	1.43E-02	1	4.00E-06	1
DroidSpan vs *Afonso*	1.43E-02	1	4.00E-06	1
DroidSpan vs *RevealDroid*	1.43E-02	1	8.51E-06	0.86
DroidSpan vs *Mudflow*	1.43E-02	1	5.84E-05	0.64

Table 6 lists the significance values and effect sizes for both the same-period and over-time settings. The results clearly show that the performance improvement of *DroidSpan* over any of these baselines was statistically significant. Further, the contrasts in effect sizes demonstrate that the improvements were significantly large: an effect size of d here means $100|d|\%$ of values in the first group (i.e., F1 of *DroidSpan*) are larger (smaller if d is negative) than values in the second group (i.e., one of the five baselines). For instance, concerning reusability, the significance was the least (with the largest p-value 4.23E-02) and the effect size was the smallest (0.75, in contrast with 1 in other comparisons) when compared to *MamaDroid*, consistent with the earlier finding that it was the best-performing baseline in terms of reusability.

Conclusion. The performance merits of *DroidSpan* over the baselines were not only noticeable in terms of sheer F1 numbers, but also statistically *significant* and *large* based on non-parametric hypothesis tests and effect-size measurement, both in same-period and over-time detection settings.

5.6 Efficiency Results

Table 7 summarizes the time spent by each technique for feature extraction (*feature time*) and ML computation (*learning time*) on each single app. In particular, the feature computation costs include all relevant time costs (e.g., for static analysis by *DroidSpan*, *Mudflow*, *RevealDroid*, and *MamaDroid*, resource analysis by *DroidSieve*, and runtime overhead by *DroidSpan* and *Afonso*). Like *Afonso*, *DroidSpan* has the total feature computation time dominated by the (5min) tracing time. For these two dynamic techniques, the cost for the ML part is quite small, though, mainly because only a small number of features were used. Nevertheless, the relatively long dynamic analysis time would impede both dynamic techniques in scenarios of *online* detection—rather, both are more suitable for offline app security analysis.

MamaDroid incurred the highest ML time because of its very-large number of features per app, which is also the cause for its high-memory demand, followed by *RevealDroid* (e.g., over 900K features for B14+M14). *DroidSieve* had relatively substantial prediction (testing) time because of the same reason (20K+ features per app). We encountered scalability issues with *Mudflow*, having failed to extract features for most of the benchmarks initially on a machine with 64GB memory (due to out of memory errors). We thus had to get a 256GB-memory machine to run it, yet still the feature extraction time was high (actually the highest among the six detectors). The reason is that

Table 7. Average time (in seconds) and storage (in KB) costs of the six techniques for each single app

	feature time	learning time	**total time**	**storage cost**
DroidSpan	351.1s	0.01s	**351.1s**	**21.2K**
MamaDroid	430.9s	41.9s	**471.9s**	**1736.01K**
DroidSieve	75.2s	3.5s	**78.7s**	**0.4KB**
Afonso	521.4s	0.015s	**521.4s**	**32.5K**
RevealDroid	78.4s	18.3s	**96.7s**	**1156.81K**
Mudflow	698.7s	0.2978s	**698.9s**	**46.67K**

Mudflow features are computed from static taint (data) flows using FlowDroid [8], which is known to be highly expensive.

In total, *Mudflow* incurred the highest cost, followed by *Afonso* and then by *MamaDroid* and *DroidSpan*. *Mudflow* took two months for 2,866 apps with 730GB memory as reported in [9]. For our study, we took almost half a year to compute the *Mudflow* features for all our benchmark apps. *DroidSieve* and *RevealDroid* are relatively lightweight among the six, especially for the feature computation part. However, their ML training part was quite expensive—just training on 10K samples for *RevealDroid*, for instance, would take over 51 hours, not counting the feature computation cost. For *MamaDroid*, the best-performing existing detector, retraining on 10K samples would take 5.5 days. These numbers echo one of our motivations for developing a sustainable detector: constant retraining might not be an option in practice.

The storage cost concerns the space (in KB) needed for storing app features, and optionally (for dynamic approaches specifically) the runtime traces. *DroidSieve* required very little storage, and two dynamic approaches incurred medium-level storage costs. *MamaDroid* and *RevealDroid* needed a substantial amount of disk space to store large feature files (again due to their great size of per-app feature vectors).

Conclusion. *DroidSpan* brings significant sustainability improvement at very reasonable time and storage costs. It is actually more efficient than the best-performing baseline (*MamaDroid*). Compared to other baselines, the much greater sustainability of *DroidSpan* should pay off the extra costs—this is particularly true because with *DroidSpan* retraining is least often needed (thanks to its greatest sustainability). *Mudflow* can face scalability challenges in practice due to its high memory demand and, even with large memory, a high analysis overhead (the highest among the six detectors).

6 CASE STUDIES

To gain a deeper understanding on how to improve the sustainability of ML classification-based malware detectors, we conducted a series of case studies to examine what it is that made *DroidSpan* a more sustainable detector compared to the baselines. More broadly, we attempted to gain intuitive insights into effective approaches to sustainable malware classification, including *DroidSpan*—while outperforming the five baselines, it still suffers from sustainability challenges (a noticeable gap remains between its performance in same-period detection and that in the over-time setting). To this end, we characterized the evolution of features of each baseline detection technique over the eight years studied, using our yearly datasets, with a focus on differences in evolutionary patterns between malware and benign apps.

Specifically, from an average-case perspective, we computed the mean feature values over all the benchmarks in each app group per detector, and then compared the group means per year as we did for *DroidSpan* as exemplified in Figure 3. Given the large number (over 5K at least) of features

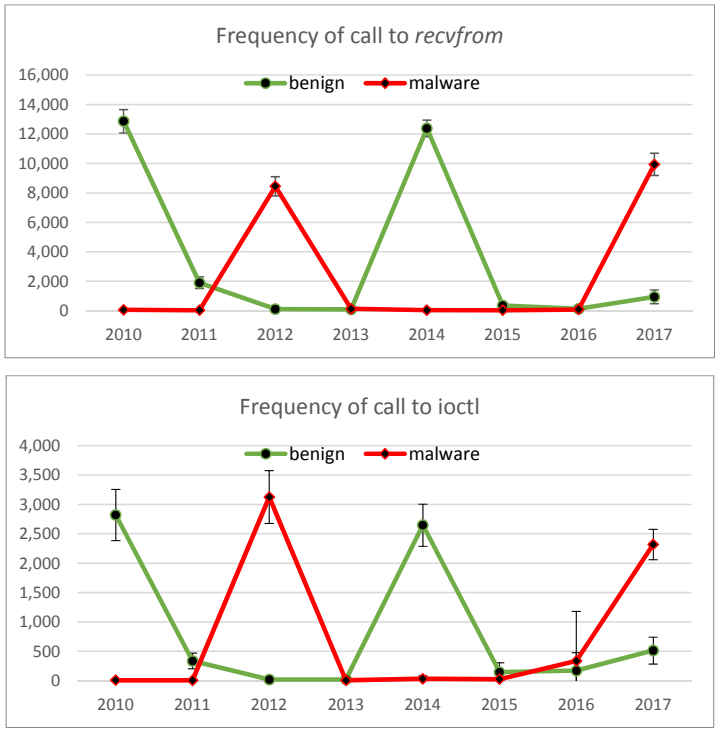

Fig. 9. Part of our case-study results: mean feature values (y axis) over years (x axis) in benign apps (blue) and malware (orange) of two *Afonso* features. Error bars show 0.95 confidence intervals of the means.

used by most of the chosen baseline techniques, we narrowed down our inspection scope to only the features that were consistently ranked at top 100 according to feature *importance* [54] in all the classification tests.

We found that for any of these five baselines, among the 100 most important features there was not even a single one that was consistently higher/lower in the malware group than the benign-app group, unlike what we observed with *DroidSpan* features. To illustrate, Figure 9 depicts the evolution of two *Afonso* features (frequency of system call `recvfrom` and that of system call `ioctl`), in the same format as Figure 3. The error bars also show Vysochanskij-Petunin confidence intervals of means (with α=0.05). As shown, there was not a *consistent* contrast (in terms of the sign) between malware and benign apps with respect to any of these features—for the same feature, the malware feature value was substantially higher than benign-app feature value in some years but the contrast was the opposite in other years. Intuitively, based on such features, the classification model trained on apps of one year was difficult to differentiate the two app groups of other years. In contrast, *DroidSpan* features were able to consistently differentiate benign apps from malware over the years (see Figure 3). Such a comparison between *DroidSpan* and the baselines provides an intuitive explanation of why *DroidSpan* achieved a higher level of sustainability. We further found that when a detector used a larger number of features with which the signs of yearly malware-benign group differences flipped frequently between adjacent years, the detector tended to have lower sustainability. For instance, *DroidSieve*, the least sustainable detector among the six studied, had the largest number of such features.

Conclusion. The sustainability of a classification-based malware detector tended to largely rely on the capability of its features in differentiating malware and benign apps consistently against the evolution of both groups. Using a larger number of features that lack this capability tended to result in a lower sustainability of the classifier.

7 THREATS TO RESULTS VALIDITY

Our results suffer from common validity threats due to benchmark selection. While we purposely collected samples from diverse sources to mitigate such threats, our yearly datasets may not be representative of the app population in respective years. Thus, our results and conclusions are best interpreted with respect to the benchmarks we analyzed. Another validity threat concerns possible errors in our tool implementations. To mitigate this threat, we maximally reused existing implementations accessible to us; for the necessary re-implementations, we did careful code review and manually validated tool correctness against a few relatively simple test samples.

Since our approach is purely dynamic, it would not work with an app that cannot be executed to produce a minimal trace that captures the exposable app behaviors for constructing a meaningful SAD profile. For instance, if an app starts with a login page that requires real, registered user account information, *DroidSpan* may not work with it because the current underlying input generator (Monkey) would most likely get stuck with the login screen. As a result, it would not be able to exercise the rest of the app hence the dynamic features would not be computed to enable accurate classification. We note that this is an implementation, rather than technical, limitation of our approach—technically, the Monkey tool we currently use can be replaced (or assisted) by a smarter input generator (e.g., [40]). Nevertheless, a minimum coverage is a constraint for the applicability of our technique. Finding out the actual minimum line coverage required for *DroidSpan* to effectively work on an app remains to be studied. Also, our findings and conclusions from the longitudinal characterization (Section 4.2) are immediately subject to the actual behaviors covered by the executions used in the study.

In addition, for constructing the SAD profile for an app, *DroidSpan* as a purely application-level approach relies on instrumenting the app to monitor the data underlying the dynamic measures that constitute the SAD profile. This necessary step has a side effect, though: the instrumentation leads to changes in both the signature and the overall behavior of the app. Fortunately, the benign/malicious nature of the app should not be much affected by these changes, as evidenced by our promising results on *DroidSpan*'s classification performance. Also, the instrumentation is only for the sake of our detection analysis—the original APK can still be used by the app user for various purposes. With respect to our experimental dataset, for the apps that were excluded due to instrumentation issues, the dominating cause was damaged/incomplete APKs. However, the changes do generally affect the applicability of *DroidSpan* itself—some apps are not amenable for instrumentation—for example, their instrumented versions may fail to run normally (e.g.,crash) as they may self-check their integrity, signature, and/or behaviors against the changes. For this and other reasons (e.g., the limitation of our instrumentation in handling some APKs), there are apps that could not be classified by *DroidSpan* because the instrumentation issues disable its feature computation. These application-level instrumentation induced limitations could be avoided by instrumenting the Android runtime system (e.g., Android ART or Dalvik VM) or using dynamic instrumentation techniques. However, these alternative approaches would bring about portability and compatibility issues: first, changing the runtime platform may not always be an option to users; second, for each different version of the runtime platform, the system-level instrumentation or other modifications to the platform may need to be redone differently with significant effort, which may not be always feasible in practice (especially in the consideration of the constant evolution of the Android ecosystem—including that of its runtime platform).

A threat to conclusion validity of our results concerns the generalizability of our methodology in the domain of mobile malware detection. The primary conclusion of this work is that the sustainability of learning-based malware detection can be improved by discovering evolution-resilient features through studying/characterizing the evolution of apps over many years. However, really stealthy malware can be purposely crafted to bypass detection by all means (e.g., evading emulator environments, self-checking integrity, etc., as discussed in more detail below in Section 8). Thus, if the features discovered are not resilient against these attacks, our methodology may still not be able to bring better sustainability for the malware detector even if the features are resilient against app evolution. On the other hand, a fully-fledged malware detector that is resilient against various adversarial attacks and capable of capturing more complex malware behaviors that we currently do not address (e.g., sensitive control flows, from sources to sinks, across threads and/or processes) may not be able to adopt our methodology to achieve better sustainability (e.g., the features used vary randomly over time, or the detector does not even use features whose evolutionary patterns can be characterized as we did with our methodology).

8 DISCUSSION

In this section, we discuss relevant concerns about *DroidSpan* as another classification-based malware detector, including its resiliency against obfuscation, sensitivity to input coverage, and limitations of the technique itself and those of its current tool implementation.

Resiliency against obfuscation. While not a focus of this work, resiliency against obfuscation is a practical concern with malware detectors [33]. Thus, we performed an additional study as a sanity check, against an obfuscated malware suite *Praguard* [41]. We used its most obfuscated subset (which adopted multiple obfuscation schemes combined: reflection, string encryption, class encryption, and renaming, etc.), of which 1,214 (of around year 2012) were usable for our study. Along with our B12 samples, this malware set was used to test each of the six detectors trained on B10+M10 and B11+M11, respectively. Our results revealed that *DroidSpan*'s resiliency was close to that of *DroidSieve* and *RevealDroid* (the two state-of-the-art obfuscation-resilient detectors)—for any of these three, the detection accuracy (F1) was only <2% lower than that when testing against B12+M12. The other three detectors saw 15–22% F1 decreases with the obfuscated malware set used. For *MamaDroid* and *Mudflow*, the reason was mainly because of their reliance on FlowDroid whose static analysis was largely impeded by code obfuscation (e.g., class/string encryption). Given its use of features based on counting Android SDK API and system calls by referring to predefined lists of function signatures, *Afonso* is directly subject to reflection and renaming obfuscation. In sum, this additional study suggested that *DroidSpan* can be expected to be overall resilient against common known obfuscation schemes. In fact, in our main experimental datasets as described in Table 2, on average over 32% of the yearly samples, in both malware and benign-app groups, adopted obfuscation schemes of some most common kinds (e.g., renaming, reflection, etc.).

Sensitivity to input coverage. Concerned about the common subjection of dynamic analysis to input availability and coverage, we originally aimed at discovering features that can capture app behaviors that are quickly exposable with random inputs and without demanding high coverage. We further investigated the impact of input coverage on the performance of *DroidSpan* by (1) comparing *DroidSpan* accuracy in testing individual samples that received varied line coverage, and (2) comparing the overall accuracy of *DroidSpan* when trained on samples traced for 5min (default) versus for shorter (1min) and longer (10min) per-app execution time. For (1), we found no correlation between the coverage for an app and *DroidSpan*'s accuracy in predicting its label. For (2), when trained on samples all traced for 1min, *DroidSpan* only had at most 1.5% drop in average F1 while increasing to 10min tracing only led to about 1% growth in F1. Thus, 5min seemed

to offer a good tradeoff. Also, the superior classification accuracy achieved by *DroidSpan* in both same-period and over-time detection settings with extensive experiments (where the line coverage of underlying traces varied widely) seems to suggest only minor dependence of *DroidSpan* on the input coverage. A main reason is that the *DroidSpan* features are all about percentages (ratios) of method calls of one kind (e.g., calls to sinks) to those of a super kind (e.g., all method calls)—as long as the ratio (i.e., feature value that contributes to differentiating benign apps from malware) can be captured, the absolute number of calls in each class traced (which is immediately affected by code coverage) did not matter much.

Sustainability against adversarial attacks. While studying sustainability of malware detectors, we essentially assumed adversarial attacks as an orthogonal problem. In practice, however, those attacks would compromise the sustainability of *DroidSpan*. Like many learning-based malware detectors, *DroidSpan* uses implementation-level features for classification. These features, once revealed, can be mimicked by malware so as to cheat the classification (e.g., a malicious app can adapt to the SAD profile of a benign app by adding glue code and/or useless methods calls). To achieve generally improved sustainability, *DroidSpan* purposely avoids using semantic features (e.g., those based on fine-grained data flows) that are explicitly relevant to malicious behaviors. Yet this strategy also facilitates adversarial attacks (e.g., code pollution). Although the primary goal of *DroidSpan* is to demonstrate a methodology for achieving sustainable app classification through app evolution studies, as a tool itself *DroidSpan* nevertheless suffers from the sustainability compromises due to such adversarial attacks.

DroidSpan could be vulnerable to advanced app obfuscation and evasion schemes particularly targeting dynamic analysis. For example, adversaries can use timing bombs to evade the detection by *DroidSpan*: once the adversarial knows that *DroidSpan* only exercises an app for 5 minutes, it can purposely delay exercising its true behaviors by more than 5 minutes. As a result, no valid dynamic features would be captured by *DroidSpan* for its detection. Other similar examples include waiting for an incoming command from a *command-and-control* server, for the satisfaction of particular system properties (e.g., the mobile device being at certain locations), and for specific user interactions (e.g., the user clicking certain widgets). *DroidSpan* currently cannot work properly with these adversarial apps.

In addition, it is well known that tricky malware can evade detection by withholding their malicious behaviors when they find being executed on an emulator. Our promising evaluation results obtained from traces collected on an emulator suggest that emulator evasion does not seem to much affect the effectiveness of *DroidSpan*. Yet our detector may still suffer from other sophisticated evasions against dynamic analysis [48]. We currently could not report the statistics on app samples that adopted those evasion schemes because detecting such schemes in a given app is not trivial—a dedicated technique would be needed for the automated detection.

Other limitations. An additional limitation of *DroidSpan* is that substantial updates of the Android SDK leading to changes of source/sink APIs may affect the performance of our technique, and thus would trigger retraining (using the updated source/sink lists). Also, although we have looked at the evolutionary patterns of malware in the past eight years and leveraged these patterns in building a more sustainable malware detector with *DroidSpan*, we cannot claim that *DroidSpan* will surely continue to work as well as shown for future malware. The unpredictable app evolution in the future would be also another trigger of retraining the *DroidSpan* classifier.

Our results on stability from Study II (Section 5.4) showed that, while substantially outperforming the five baseline techniques, *DroidSpan* still has large room for improvement in terms of its absolute classification accuracy (72% F1) in the over-time detection setting. One possible contributing reason is that some benign apps might have been falsely classified as malware because they had abnormal

feature values in terms of sensitive accesses (e.g., abnormally higher frequency of source/sink calls). As partially depicted in Figure 3, our *DroidSpan* features were selected because they can separate benign apps and malware consistently over time in an *average-case* analysis—for some outlier apps, the feature values might have been closer to malware than to benign apps. As a result, these outlier apps can be falsely classified (i.e., resulting in false positives). Meanwhile, there might also have been individual malicious apps that were falsely classified as benign (resulting in false negatives) because their values of certain features were closer to benign apps than to malware. In addition, these kinds of abnormal cases can also happen due to the diversity within each group (e.g., benign apps of different functionality categories, and malware of different families).

9 RELATED WORK

Characterization of Android apps. Previous characterization studies of Android apps mostly focused on (*static*) metrics extracted from the app code. For example, in [24], 1,100 popular apps were studied to understand their use and misuse of private information of mobile phones and users through reverse engineering. A few *dynamic* characterization studies exist, which concern the installation and activation methods of malware only [66] or execution structure of benign apps only [18]. Another examples include the dynamic study [61] that profiles apps in terms of their inter-component communications (ICC) [36] and network traffic. CopperDroid [56] can be used for characterizing system calls in apps, and the recently developed toolkit in [16, 17] serves both static and dynamic characterizations. In contrast, our characterization is purely dynamic, focusing on run-time sensitive accesses in both benign and malicious apps. More importantly, our characterization aims at a longitudinal examination of the evolution of Android apps. The characterization of evaluation datasets in [42] reveals the evolutionary characteristics of both malware and benign apps also, but it instead focuses on API calls in apps and is static.

Android Malware Detection. Numerous approaches have been proposed for detecting Android malware, by *statically* analyzing data and/or control flows [26, 32, 58, 64], API use [2, 9, 62, 64, 65], app descriptions [31], and/or installation-time permission [7, 9, 19, 51, 62]. ICCDetector [63] distinguishes malware from benign apps based on their different patterns in ICCs. The five state-of-the-art malware detectors have been discussed in detail earlier.

Dynamic malware detectors have mainly exploited monitoring system and/or API calls in apps [3, 10, 19], or high-level behaviors exhibited through the usage of system resources such as file and network access [52]. Some of these approaches used static features in addition to dynamic ones [19, 39, 51]. Approaches relied on static calls in app code are generally vulnerable to code obfuscation (e.g., reflection) [54]. As suggested by the study in [42], prior approaches are rarely shown to be capable of sustainable detection. For example, DroidAPIMiner [2], which was used as the baseline of MamaDroid, did not sustain its capabilities up to one year without retraining [42]. *DroidSpan* targets a better sustainability of malware detection than MamaDroid and other state-of-the-art alternatives, using only dynamic features based on the extent and distribution of *exercised* sensitive accesses and vulnerable, method-level *control flows* in app executions.

As a static detector, Mudflow [9] also uses features based on source/sink usage like our approach. It builds a normal behavior model based on the statement-level sensitive *data flows* [8, 13] in a training set of benign apps and absolute numbers (of flow paths) to identify new malware, compared to our approach using dynamic method-level control flows and relative statistics (percentages). Our empirical results suggest that, while by eliminating reliance on new malware samples for training Mudflow achieved relatively consistent classification performance across same-period and over-time settings, it suffered from low accuracy partially because the fine-grained data-flow features it relied on varied largely within both malware and benign apps (hence the constant lack

of consistently substantial differences between the two groups in terms of those features). Another static detector [53] leverages common API-call subsequences of source-sink information flow paths (referred to as complex flows) to model app behaviors and then detects malware from benign apps based on their differences in terms of complex-flow patterns (characterized using n-gram). Given its behavioral modeling based on API sequences, this approach is expected to suffer from sustainability barriers due to the rapid API evolution in the Android ecosystem.

Researchers have approached the sustainability problem of malware detectors through adaptive classification via online/active learning [43, 46]. Also, detecting concept drift in malware classifiers [37] helps understand and diagnose the problem hence informs the design of more sustainable malware detectors [5]. Sharing a similar goal to ours, these techniques can be leveraged to further enhance the sustainable detection performance of our approach (e.g., by tuning the classifier to delay time decay [46]). A preliminary version of this work [11] only compared *DroidSpan* to *MamaDroid* against smaller yearly datasets for a shorter maximum span, and a summary of the major results and conclusions was presented in [12]. This paper also extended our most recent version of *DroidSpan*, which was only summarized very briefly in [27], mainly by including *Mudflow* in the comparative study.

10 CONCLUSION

As malware remains rampant in the Android ecosystem threatening its large user base, defending against Android malware is crucial. Numerous approaches have been proposed, mostly training a learning model to predict the label of novel apps. Yet, existing approaches tend not to sustain without retraining, which is not practical for detecting emerging malware.

To understand this problem, we performed extensive studies on how representative classification-based malware detectors for Android sustain their performance over time. We found that state-of-the-art detectors generally suffer from undesirable sustainability, and as a main cause they used features that do not well differentiate benign apps from malware against evolution of both groups. Leveraging our study findings, we developed *DroidSpan*, a novel malware detection approach based on a new behavior profile of Android apps that models the distribution of their sensitive accesses. We also performed a longitudinal characterization using this profile, and revealed that our studied benign apps and malware were *consistently* separable in terms of the profile over the past eight years. We quantitatively defined sustainability as *a new validity factor* of a learning-based app classifier and evaluated it on *DroidSpan* against five state-of-the-art malware detectors covering representative static and dynamic approaches. Our results showed that *DroidSpan* outperformed the five baselines in sustainability with statistically significant and large improvements at reasonable time and storage costs. By grounding the superior sustainability in the use of features consistently separating malware and benign apps, *DroidSpan* demonstrated a methodology for designing a sustainable malware detector.

11 ACKNOWLEDGMENTS

We would thank the reviewers for their constructive and insightful comments, which helped us improve the original manuscript significantly. This work was supported in part by the National Science Foundation grant CCF-1936522. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsor.

REFERENCES

[1] 2015. Android malware accounts for 97% of all malicious mobile apps. https://www.theinquirer.net/inquirer/news/2414949/android-accounts-for-97-percent-of-all-mobile-malware.

[2] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*. 86–103.

[3] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. 2015. Identifying Android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques* 11, 1 (2015), 9–17.

[4] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of Working Conference on Mining Software Repositories*. 468–471.

[5] Paulo RL Almeida, Luiz S Oliveira, Alceu S Britto Jr, and Robert Sabourin. 2018. Adapting dynamic classifier selection for concept drift. *Expert Systems with Applications* 104 (2018), 67–85.

[6] Nicoló Andronio, Stefano Zanero, and Federico Maggi. 2015. Heldroid: Dissecting and detecting mobile ransomware. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, 382–404.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of Network and Distributed System Security Symposium*.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of ACM Conference on Programming Language Design and Implementation*. 259–269.

[9] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 426–436.

[10] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of ACM workshop on Security and privacy in smartphones and mobile devices*. 15–26.

[11] Haipeng Cai. 2018. A Preliminary Study On the Sustainability of Android Malware Detection. *arXiv preprint arXiv:1807.08221* (2018).

[12] Haipeng Cai and John Jenkins. 2018. Towards sustainable Android malware detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 350–351.

[13] Haipeng Cai and John Jinkens. 2018. Leveraging Historical Versions of Android Apps for Efficient and Precise Taint Analysis. In *IEEE/ACM Working Conference on Mining Software Repository (MSR)*. 265–269. [https://dl.acm.org/authorize?N661846pdf, https://bitbucket.org/sabatu/iterative-taint-analysisproject].

[14] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2018. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security* 14, 6 (2018), 1455–1470.

[15] Haipeng Cai and Barbara Ryder. 2016. *Understanding application behaviours for android security: A systematic characterization*. Technical Report. Department of Computer Science, Virginia Polytechnic Institute & State Unversity.

[16] Haipeng Cai and Barbara Ryder. 2017. Artifacts for Dynamic Analysis of Android Apps. In *International Conference on Software Maintenance and Evolution (ICSME)*. 659.

[17] Haipeng Cai and Barbara Ryder. 2017. DroidFax: A toolkit for systematic characterization of Android applications. In *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*. 643–647.

[18] Haipeng Cai and Barbara Ryder. 2017. Understanding Android Application Programming and Security: A Dynamic Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. 364–375.

[19] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware. In *Proceedings of ACM Asia Conference on Computer and Communications Security*. 377–388.

[20] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 429–440.

[21] Norman Cliff. 1996. *Ordinal methods for behavioral data analysis*. Psychology Press.

[22] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. DroidScribe: Classifying Android Malware Based on Runtime Behavior. *Proceedings of IEEE Mobile Security Technologies* (2016).

[23] Karim Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara Ryder. 2018. Identifying Mobile Inter-App Communication Risks. *IEEE Transactions on Mobile Computing* (2018).

[24] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*. 21–21.

[25] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2015. Android security: a survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials* 17, 2 (2015), 998–1022.

[26] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis. In *FSE*.

[27] Xiaoqin Fu and Haipeng Cai. 2019. On the deterioration of learning-based malware detectors for Android. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 272–273.

[28] Hisham Shehata Galal, Yousef Bassyouni Mahdy, and Mohammed Ali Atiea. 2015. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques* (2015), 1–9.

[29] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of Android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 11.

[30] Google. 2017. Android Monkey. http://developer.android.com/tools/help/monkey.html.

[31] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1025–1035.

[32] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day Android malware detection. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services*. 281–294.

[33] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 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*. ACM, 421–431.

[34] https://virusshare.com/. 2016. VirusShare. https://www.virusshare.com/.

[35] John Jinkens and Haipeng Cai. 2017. Dissecting Android Inter-Component Communications via Interactive Visual Explorations. In *International Conference on Software Maintenance and Evolution (ICSME)*. 519–523.

[36] John Jinkens and Haipeng Cai. 2018. ICC-Inspect: Supporting Runtime Inspection of Android Inter-Component Communications. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft)*. 80–83.

[37] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 625–642.

[38] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *Proceedings of USENIX Security Symposium*. 351–366.

[39] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. 2015. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proceedings of IEEE Computer Software and Applications Conference*, Vol. 2. 422–433.

[40] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 643–653.

[41] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security* 51 (2015), 16–31.

[42] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of Network and Distributed System Security Symposium*.

[43] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. 2017. Context-aware, adaptive, and scalable Android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (2017), 157–175.

[44] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test*. ACM, 34–37.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[46] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security Symposium*.

[47] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.. In *Proceedings of Network and Distributed System Security Symposium*.

[48] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of IEEE/ACM International Conference on Software Engineering*.

[49] International Data Corporation (IDC) Research. 2016. Android dominating mobile market. http://www.idc.com/promo/smartphone-market-share/.
[50] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.
[51] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based Android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing* (2016).
[52] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": A Behavioral Malware Detection Framework for Android Devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190.
[53] Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steven Y Ko, and Lukasz Ziarek. 2018. Android malware detection using complex-flows. *IEEE Transactions on Mobile Computing* 18, 6 (2018), 1231–1245.
[54] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and accurate classification of obfuscated Android malware. In *Proceedings of ACM Conference on Data and Application Security and Privacy*. 309–320.
[55] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of Android malware and Android analysis techniques. *Comput. Surveys* 49, 4 (2017), 76.
[56] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *Proceedings of Network and Distributed System Security Symposium*.
[57] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. 2015. Securing Android: a survey, taxonomy, and challenges. *Comput. Surveys* 47, 4 (2015), 1–45.
[58] Omer Tripp, Marco Pistoia, Pietro Ferrara, and Julia Rubin. 2016. Pinpointing Mobile Malware Using Code Analysis. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. 275–276.
[59] D.F. Vysochanskij and Y.I. Petunin. 1980. Justification of the three-sigma rule for unimodal distributions. *Theory of Probability and Mathematical Statistics* 21 (1980), 25–36.
[60] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying E. Ye. 2011. *Probability and Statistics for Engineers and Scientists*. Prentice Hall.
[61] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. ProfileDroid: multi-layer profiling of Android applications. In *Proceedings of ACM International Conference on Mobile Computing and Networking*. 137–148.
[62] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. Droidmat: Android malware detection through manifest and API calls tracing. In *Proceedings of Asia Joint Conference on Information Security*. 62–69.
[63] Ke Xu, Yingjiu Li, and Robert H Deng. 2016. ICCDetector: ICC-Based Malware Detection on Android. *IEEE Transactions on Information Forensics and Security* 11, 6 (2016), 1252–1264.
[64] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *Proceedings of European Symposium on Research in Computer Security*. 163–182.
[65] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware Android malware classification using weighted contextual api dependency graphs. In *Proceedings of ACM Conference on Computer and Communications Security*. 1105–1116.
[66] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of IEEE Symposium on Security and Privacy*. 95–109.