MAST: Triage for Market-scale Mobile Malware Analysis

Saurabh Chakradeo, Bradley Reaves, Patrick Traynor
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
{schakradeo, brad.reaves}@gatech.edu, traynor@cc.gatech.edu

William Enck
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
enck@cs.ncsu.edu

ABSTRACT

Malware is a pressing concern for mobile application market operators. While current mitigation techniques are keeping pace with the relatively infrequent presence of malicious code, the rapidly increasing rate of application development makes manual and resource-intensive automated analysis costly at market-scale. To address this resource imbalance, we present the Mobile Application Security Triage (MAST) architecture, a tool that helps to direct scarce malware analysis resources towards the applications with the greatest potential to exhibit malicious behavior. MAST analyzes attributes extracted from just the application package using Multiple Correspondence Analysis (MCA), a statistical method that measures the correlation between multiple categorical (i.e., qualitative) data. We train MAST using over 15,000 applications from Google Play and a dataset of 732 known-malicious applications. We then use MAST to perform triage on three third-party markets of different size and malware composition—36,710 applications in total. Our experiments show that MAST is both effective and performant. Using MAST ordered ranking, malware-analysis tools can find 95% of malware at the cost of analyzing 13% of the non-malicious applications on average across multiple markets, and MAST triage processes markets in less than a quarter of the time required to perform signature detection. More importantly, we show that successful triage can dramatically reduce the costs of removing malicious applications from markets.

Keywords

Mobile application security, Triage, Multiple correspondence analysis

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

1. INTRODUCTION

Application markets have simplified the process of finding and installing software on smartphones, creating an efficient channel between developers and end users [4, 8, 21, 51]. Unfortunately, they have also provided attackers with an easy point of entry into mobile devices in the form of vulnerable and malicious applications [13, 25–27, 32, 34, 35]. Some markets have responded thus far with a variety of proactive and reactive approaches to this problem (e.g., Apple's manual analysis and Google's Bouncer [31]). Though these approaches have thus far minimized the amount of malware that appears on their respective markets, many alternative Android markets [3,20,49] still remain completely unprotected against malware. Also, the time and monetary cost as well as the need for in-depth inspection of the applications will rise as malware takes greater lengths to avoid detection, and the number of new samples to analyze continues to increase as developers create new and update existing applications. To continue to be effective and work within malware detection budgets, market providers, antivirus companies, and independent researchers must then, more than ever, strategically spend manual effort and computationally expensive program analysis.

Mobile application security is not the first discipline to wrestle with a mismatch of resources and duties. Medical facilities regularly perform triage, or the prioritization of limited resources based on the perceived condition of each individual within the population of patients. Triage is neither diagnosis nor treatment. Rather, it allows medical personnel to immediately deal with patients with the greatest obvious needs while delaying or dismissing treatment for others. Developing such perception and prioritization in the mobile application space, where most applications are in fact benign, would allow scarce resources to be dedicated to the investigation of applications that have the greatest potential to be dangerous.

In this paper, we present the Mobile Application Security Triage (MAST) infrastructure. MAST develops a perception of suspicion for applications through the use of Multiple Correspondence Analysis (MCA) [1], a technique used in the social sciences to determine the statistical correlation between multiple categorical (i.e., qualitative) data. In particular, we develop a "questionnaire" for Android applications that looks for strong relationships between *declared* indicators of application functionality (e.g., permissions, intent filters, the presence of native code, etc.) given the key insight that *these declared indicators are required for malware to perform its malicious functionality.* Effectively, MAST operates under the assumption that the configurations of declared indicators of legitimate app are distinct from malware, which MCA identifies as outliers in a population.

Our methodology is developed using over 15,000 applications from Google Play [21], the Contagio mobile malware repository [43] and malicious applications found by Zhou et al. [53]. We then apply MAST to three third-party application markets: Anzhi (28,760 apps) [3], Ndoo (4,324 apps) [39], and SoftAndroid (3,626 apps) [49].

WiSec'13, April 17–19, 2013, Budapest, Hungary.

Our results show that MAST dramatically reduces the effort spent on non-malicious applications. Using MAST, existing detection tools can identify on average 95% of malware in third-party markets at the cost of analyzing 13% of the non-malicious applications in them. As a side-effect of performing this triage, we discovered widespread misuse of the Android default application signing key, appearing in 1,672 applications across the Android Market sample, the malware dataset, and the three third-party markets.

We make the following contributions:

- **Develop the MCA-based MAST Architecture:** As the rate with which mobile apps are added to markets increases, performing deep security analysis of those apps will become a "big data" problem. We develop an infrastructure for directing scarce analysis resources to address this problem. MAST uses MCA to rank applications with a degree of suspicion. Although implemented for Android, this architecture is reusable in any system in which required qualitative declarations of application functionality are available, and is extensible to accommodate new classes of malicious applications.
- **Analyze multiple third-party Android markets:** We train and evaluate the effectiveness of MAST using a corpus of more than 50,000 Android applications from a range of different markets. Even with the diversity in size and malware population of these markets, MAST effectively triages malicious apps—on average 95% of malware present in the market can be detected at the cost of analyzing 13% of the non-malicious applications.
- **Generate rankings faster than any lightweight analysis:** MAST is designed to direct resource intensive operations (e.g., manual analysis), so generating these rankings requires less time than even the most lightweight analysis tools. Specifically, our MAST infrastructure ranks applications more than 4 times faster than signature detection.

MAST is a comprehensive yet lightweight mechanism for identifying malware in Android. Prior work has used rules or simple statistical measures to detect malware [16, 19, 22, 54], while more advanced analyses [44, 47] have reported worse or comparable efficacy, while failing to characterize their efficiency and efficacy on markets with actual threats. The MAST architecture is fundamentally more flexible and robust than a rule-based filtering scheme, and this paper is the first to demonstrate that advanced techniques can be useful in practice.

Finally, MAST is not designed as a replacement for manual analysis (e.g., as in Apple's App Store) or automated malware detection tools (e.g., Google's Bouncer [31]). Rather, *MAST provides a rank-ordered list over which additional, more heavyweight, techniques can be applied.* Alternative Android markets, that have human resources to manually analyze only a small subset of applications, can use MAST to decide which applications deserve the most attention. In the case that markets want to scan all the applications, MAST can aid in deciding which applications require deep, costly analysis and which ones require just a cursory anti-virus scan. The necessity of properly allocating malware analysis resources is going to become evident as smartphone malware authors start adopting techniques such as polymorphic and obfuscated code and malware researchers respond with complex static and dynamic analysis techniques. The goal of MAST is to robustly direct malware analysis resources in a manner that is not dependent on any one characteristic and is thereby more resilient to the "hide-and-seek" games of malware authors. Such an approach is valuable not only to large market operators with extensive resources, but also to smaller markets and even academic and industry researchers as well.

2. RELATED WORK

Application markets currently rely on both proactive and reactive mechanisms for dealing with mobile malware. The proactive approach typically relies on the use of automated tools to detect vulnerabilities. Simple approaches are evadable [38, 42], and complex analyses can become unsustainable as the complexity and number of applications requiring certification grows. Worse still, many proactive analyses require human intervention [37], making them fundamentally unsuitable at market scale. Malicious applications have already exploited this fact and have been seen in multiple markets [25–27, 29]. Reactive approaches recognize this and attempt to recover from such infections. Market controlled "kill-switches" [7, 10] allow malicious apps to be removed post infection; however, significant damage may have already been done.

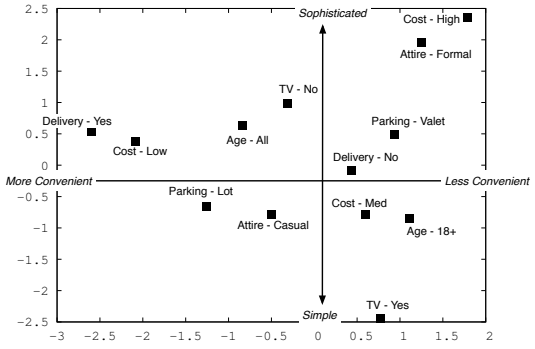
Various techniques have been proposed to improve security by reducing vulnerabilities in applications [12, 15, 18] and creating stronger detection mechanisms for malware and grayware [13, 15, 16, 23, 55]. Traditional antivirus products [33, 40] use known malware signatures to detect malicious applications; however, they fail to capture new threats. Enck et al. [14] use taint-tracking to detect information flow between sources of private information (e.g. IMEI, phone number) and external data sinks (e.g. internet, SMS). High-level run-time behavior [9] and power [28, 30] have also been explored for malware detection. However, when automating dynamic analysis, it is nontrivial and costly to ensure complete coverage to trigger malicious payloads.

A number of efforts rely on Android permissions to determine the maliciousness of applications. Kirin [16] uses static, conjunctive rule sets to define possible malicious behaviors and warn the user at install time. Barrera et al. [6] use self-organizing maps to analyze permission usage patterns in applications. Felt et al. [19] use permission counts and a comparison of individual permissions to reason about malicious applications. However, all these approaches suffer from high false positive rates and are fundamentally limited by their inability to fully analyze the correlation between high dimensional data associated with applications. DroidRanger [54] and RiskRanker [22] are closer in spirit to MAST. DroidRanger's permission-based filtering had a false positive rate of 40%, after which Zhou et al. [54] had to manually analyze and then add checks for specific Android intents and native code to reduce the false positive rates. Further, DroidRanger does not provide a generalized easily reproducible methodology for selecting such "expert" features. RiskRanker, despite its name, only performs classification into three categories: high, medium, and low risk. Apps are classified as high-risk if they match vulnerability-specific signatures (e.g., root exploits), and medium-risk if they use SMS without a code-path to an *onClick()* callback. MAST improves upon both DroidRanger and RiskRanker by providing a statistical foundation for allocating valuable malware analysis resources based on 208 behavioral features.

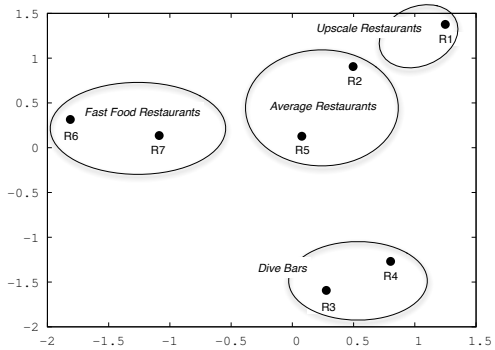
The closest research to MAST is concurrent and independent work done by Sarma et al. [47] and Peng et al. [44]. These works test several techniques (but not MCA) over application permissions to provide rankings of relative risks *to users*. Peng et al. show superior results to Sarma et al. Their goals include low user warning rates and developer accessibility; by contrast, MAST uses not only permissions, but application intents and the presence of native code to determine suspicion *for analysts*. It is a strict design goal in [44] that developers understand how to reduce their risk; MAST does not need this property because it is targeted to analysts, not developers. However, *our results are better representative of triage in practice* because we test against third party markets with malware that are unrelated to our training data. Peng et al. train and test

Table 1: A Sample Restaurant Data Set for MCA

Restaurant	Cost	Parking	Attire	Ages	Delivery	Television
R1	High	Valet	Formal	18+	No	No
R2	Med	Valet	Formal	All	No	No
R3	Med	Lot	Casual	18+	No	Yes
R4	Med	Valet	Casual	18+	No	Yes
R5	Med	Valet	Casual	All	No	No
R6	Low	Lot	Casual	All	Yes	No
R7	Low	Lot	Casual	All	No	No



(a) Correlations between restaurant properties (column similarity)



(b) Similar restaurants clustered together (row similarity)

Figure 1: Output of MCA for the example restaurant data

using only the official Google Android market and external malware datasets. We also provide measurement of our run time performance to show that MAST is practical for smaller markets and individual researchers.

Scalability and effective analysis of correlation between multiple attributes are important factors to consider when selecting such "expert" features. Finding these has been a problem in PC malware analysis as well. McBoost [45] uses n-gram analysis and Multi-Layer Perceptron on executable heuristics to detect packed code. ForeCast [41] predicts the information gain of analyzing an executable using a modified linear classifier fed with static executable features and behavioral features from dynamic analysis. BitShred [24] uses feature hashing and co-clustering to enable fast extraction of information from malware samples. However, though these techniques perform well for PC malware, they are not built to handle the nominal categorical data available from mobile applications. On the contrary, Multiple Correspondence Analysis (MCA) [1] is designed to analyze the correlation between multiple categorical attributes. Hence, we design MAST to leverage MCA and provide a lightweight triage technique to narrow down the search space of applications for resource-intensive analyses.

3. MCA BACKGROUND

Multiple Correspondence Analysis (MCA) is an analysis technique used to illuminate the relationships in a dataset with categorical variables. We believe this technique to be more appropriate than generic machine learning techniques (e.g., SVM[1]) as MCA is specifically designed to deal with the categorical data that describes apps. To help the reader understand MCA and gain an intuition for why we use it, we describe an example MCA applied to restaurants and then provide a high-level description of how MCA works. A more rigorous description of MCA is provided in the appendix.

3.1 Example Analysis

Table 1 shows the results of a questionnaire given to seven hypothetical restaurants. In MCA terminology, each restaurant is an "individual", the categorical variables that describe a restaurant are "questions," the values of questions for a given individual are "answers," and the set of questions is termed a "questionnaire." These terms derive from MCA's chief application in the social sciences.

For a given dataset, MCA maps each individual and answer into a set of coordinates in "principal axes." Principal axes condense the information contained in the data sets so that the majority of the information is reflected in only a few axes. The individuals' coordinates in principal axes are scaled in magnitude to reflect how unique their answers are and placed so that the variance of individuals' positions in an axis is maximized. Answers' coordinates in principal axes are scaled and placed based on the individuals that give a particular answer. Related individuals and questions are plotted near each other in the space of principal axes.

Figure 1a and Figure 1b show plots of the answers (characteristics of the restaurants) and the first two principal axes of individuals (restaurants) resulting from the MCA analysis of the data. The left side of Figure 1a shows that low-cost restaurants often offer delivery and are family-friendly; the presence of delivery towards the outside of the plot indicates that it is an unusual feature of a restaurant, while restaurants with casual attire tend to be more common. Likewise, high-cost restaurants are less common, and are strongly correlated with formal attire in this sample (found in the top right corner of Figure 1a). From inspection of Figure 1b, restaurant R1 is a clear outlier as the most unique restaurant in the set.

Principal axes can act as indicators of "hidden variables" that better describe the collection of answers given by an individual. One common interpretation technique in an MCA analysis is to attempt to describe the principal axes of the resulting plot to determine these hidden variables. This can only be done by a human analyst, as the description can be highly subjective. In Figure 1a we have provided an interpretation of the axes. While we display restaurants and their characteristics in separate plots for clarity, both restaurants and characteristics are plotted on the same scales on the same principal axes, so an interpretation of an axis in one plot is valid for the other plot. The horizontal axis in the plots reflects *convenience*; restaurants and characteristics to the left of the vertical axis tend to be cheaper and more accessible, while those to the right of the axis tend to require more effort to enjoy. The vertical axis in the plot reflects *ambiance*; restaurants and characteristics below the horizontal axis tend to be simpler, while those above the horizontal axis tend to more sophisticated. Axis descriptions come from consideration of both restaurants and characteristics and their locations with respect to the principal axes. Thus, we finally are able to say that restaurants R1 is an "upscale restaurant," R2 and R5 are "average restaurants," R3 and R4 are "dive bars," and R6 and R7 would be best classified as "inexpensive fast-food restaurants". This classification is illustrated in Figure 1b.

An alternative to attempting to describe the axes is to compute a distance from the mean point for each individual; this approach is

[1]In fact, Sarma et al. [47] use SVM and are less effective at ranking malware than MAST.

used by MAST to avoid the need for semantic description. Had we computed a distance from the center for each restaurant, it would be apparent without examining the plots that R1 is the most unique.

3.2 Informal Description of MCA

This section describes the mathematics of MCA in more general terms, and it follows from and is inspired by Le Roux et al. [46]. First, we consider a set of individuals described by their answers to N questions. If there were only two questions, ($N = 2$), all individuals could be plotted in a two-dimensional plane based on values of these questions. Similarly, individuals described by three questions could be plotted as a "cloud" of points in three-dimensional space. In the case of four or more questions, similar constructions exist for higher dimensions, even if these are hard to visualize.

If one needed to compress the information of the cloud, one could project the cloud into only a few dimensions. A projection can be conceptualized as the shadow that a cloud would cast onto a lower-dimensional space. For example, a three-dimensional cloud of points illuminated from above the cloud would cast a shadow onto a plane. Similarly, that plane can be treated as cloud and projected onto a single line; conceptually, this is like looking at only the x-coordinates of points in a plane.

Given a cloud with large N, certain dimensions frequently provide more insight about the data than others. Often, questions are redundant or strongly correlated, and they only hide more interesting data. A problem in analyzing a large cloud is that the interesting dimensions may not be known *a priori*. In fact, this may be the goal of the analysis. To address this problem, MCA transforms a cloud from a nominal set of N-dimensional coordinates into principal coordinates. Principal coordinates describe a coordinate frame where the first dimension is guaranteed to show more information than the second dimension, the second to show more information than the third, and so on. These new dimensions are termed *principal axes*, and coordinates in these axes are termed *principal coordinates*. Because a majority of the information will be in the first few principal axes, only the first principal axes will need analysis.

MCA uses two insights for transforming a cloud into principal coordinates: a) scaling less-common values to be more distinct than more-common values and b) variance of the data. The process of transforming a cloud from its natural coordinates into principal coordinates (coordinates in terms of the principal axes) consists of two logical steps. In the first step, MCA scales each coordinate in each nominal axis by the probability of the coordinate value being chosen by an individual. This first step skews the shape of the cloud so that uncommon values are further from the origin than common values. The second step starts by projecting the cloud onto a single line. MCA then rotates the cloud in N-dimensions until the variance of the *projection* along the line is maximized. This line becomes the first principal axis. Next, MCA defines a new axis orthogonal to the first, and rotates the cloud in $(N - 1)$-dimensions to maximize the variance on the new axis while keeping the variance on the first axis unchanged. MCA continues this process until N principal axes are defined. Once all axes are defined, MCA describes all points in the cloud in terms of coordinates of each of the principal axes. By construction, the principal axes are in order of decreasing variance.

The result of MCA is a new set of coordinates in N principal axes. These allow for a selection of the most important principal axes to be plotted, analyzed, and interpreted.

4. METHODOLOGY

MAST uses MCA to rank applications. The resulting ranking provides an ordering of relative suspicion. Application security teams can use the MAST ranking to effectively allocate scarce resources (e.g., manual code reviews, automated static program analysis or dynamic analysis).

We develop the MAST architecture based loosely on the principles of boosting [48], a machine learning meta-algorithm. Boosting aims to improve the accuracy of any learning algorithm by creating numerous rough and moderately accurate weak classifiers and combining their results to get an accurate strong learner. Even though the weak classifiers individually are not highly accurate, the merging of their results generates an accurate "boosted" algorithm. MAST creates these rough indicators of suspiciousness by running MCA on multiple subsets of applications. For clarity, we call each subset a "poll." Polls are selected using characteristics of malicious behavior, thereby allowing MCA to identify outliers for that characteristic. Boosting weighs each weak rule depending on its performance. MAST uses a binary weight system — the polls we select have weight one, and all others have weight zero. MAST then merges the results of the individual polls to determine a total ranking that represents a relative degree of suspicion.

Figure 2 shows the MAST architecture. The first step (Section 4.1) identifies application attributes that define interesting security properties. The second step (Section 4.3) combines related sets of attributes to create MCA questions (Section 4.2). The third step runs MCA over multiple polls to generate rough indicators of suspicious behavior. The final merge step (Section 4.4) combines these rough indicators to create an accurate MAST ranking of application suspiciousness.

4.1 Attribute Identification

MAST is designed to be less costly than deep analysis techniques. Therefore, attributes must be easy to obtain from the application package (as opposed to the result of code recovery). We look at Android's package manifest and simple information about files in the package. We chose not to use market-specific metadata such as categories, user ratings, and descriptions, because we do not want to tie MAST to any specific market. That said, deployments of MAST might wish to incorporate market metadata as appropriate. We consider permissions, intent filters, native code and presence of zip files.

Permissions: Android uses permissions to restrict access to security sensitive operations (e.g., sending SMS messages, reading location from GPS, and placing emergency calls). In order to access these resources, the application's developer must declare the corresponding permission in the manifest file. Permissions are only granted at install-time and cannot be changed without upgrading the application. There are several conditions under which permissions are granted (e.g., based on package signatures). However, for the purposes of this paper, the reader may assume all permissions are granted if the user elects to install the application. Interested readers are referred to prior discussions of Android security for additional details [11, 17]. Finally, third-party application developers can define new permissions. MAST currently uses the 114 permissions defined by the Android framework; however, custom permissions can be added if needed.

Intent Filters: Android uses intent messages for interprocess communication (IPC). Intents provide interfaces to core platform functionality as well as interactions between third-party applications. Frequently, intents are addressed to "action strings." Depending on the type of intent, the Android middleware uses action strings to notify applications of the event or resolve the best application for the task. An application indicates in its manifest file the ability to handle an intent by specifying an *intent filter* for a pre-agreed upon

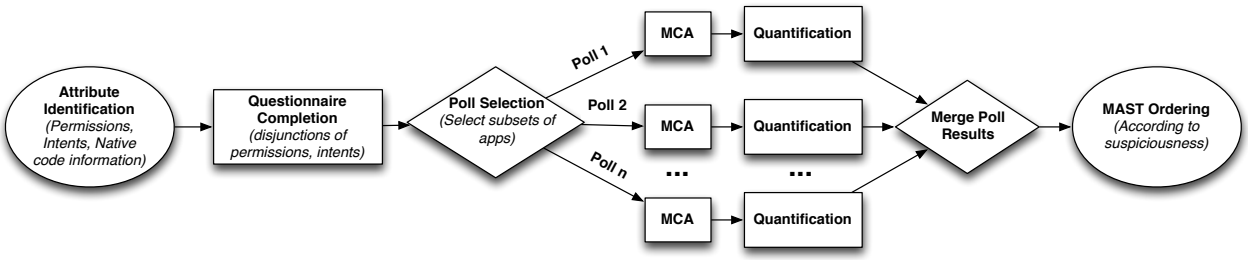


Figure 2: The MAST architecture: Building an analysis architecture using MCA

action string.[2] Intent filters provide a pluggable architecture that allows OEMs and third-party developers to customize the user experience. However, applications can also abuse this extensibility by handling events and tasks in ways that harm the user. MAST currently uses the 92 action strings defined by the Android framework. Similar to permissions, this knowledge base can be trivially extended. Finally, intent filters have an optional *priority* field that influences the order in which Android delivers intents to applications. By defining a high priority intent filter, an application may be able to cancel the intent before other applications receive it. We classify an application's intent filter for each action string as *priority*, *default*, or *none*. For the purposes of our evaluation, we consider an intent filter to be *priority* if it specifies the priority field.

Native Code: Android applications are primarily written in Java; however, the native development toolkit (NDK) allows third-party application developers to include native libraries. While native libraries provide valuable performance benefits for computationally bound applications (e.g., games), they also have been used by malware to exploit root vulnerabilities (e.g., DroidDream). We classify an application based on whether or not it includes native libraries. To identify native libraries, we search the `.apk` archive for files with the native library magic number using `libmagic`. Note that we make no attempt to identify downloaded libraries. Doing so statically would require program analysis, which is computationally too expensive for triage.

Zip Files: Android applications are distributed as `.apk` archive files. Archiving reduces the amount of data that needs to be downloaded when installing an application. However, as no restrictions are put on the type of data these application archives can contain, they have been used to carry malicious payloads as zip files (e.g., BaseBridge carries an entire malicious application as its payload). As a final attribute, we classify an application based on the presence or absence of zip files inside the main application archive. We do not recursively analyze the contents of the zip file.

4.2 MCA Questionnaire

MAST carefully defines an MCA questionnaire to aid security triage. We identified many different attributes: 114 permissions, 92 intent types (with and without priority), the existence of native code, and the presence of zip files. Creating a question for each attribute produces a very high dimensional categorical data set with relatively limited interrelationships. In practice, this results in the "being unique is common" phenomenon.

We considered two methods of combining attributes: conjunctive questions and disjunctive questions. A conjunctive question is true if an app has *all* attributes in a set. Similarly, a disjunctive question is true if an app has *at least one* of the attributes in a set.

Enck et al. [16] define nine conjunctive questions for their Kirin system. These questions (called rules) are the results of a malware oriented security requirements engineering of the Android platform. As we show in Section 5, these rules do not perform as well as the disjunctive questions used in MAST.

MAST uses disjunctive questions to collapse attributes into more general descriptions of functionality. For example, when identifying malware, it is often sufficient to know that an application has permission to perform an SMS related operation as opposed to the specific types of SMS operations. From the MCA perspective, a disjunctive question increases the likelihood that two applications have a property in common, and therefore clusters applications based on their functionality.

The questionnaire consists of permission questions, intent filter questions and native code and zip file questions. The questions group attributes by functionality. Permission, native code, and zip file questions are true or false questions. For example, the answer to the "SMS" question is true if an application requests at least one of the SMS-related permissions. In contrast, the intent filter questions have possible answers of "priority," "default," and "none." An answer of "priority" indicates that the application has at least one intent filter for a listed action string that defines the priority field. If the application does not have any matching priority intent filters, but it does have an intent filter for one of the listed action strings, the answer is "default." Otherwise, the answer is "none."

Table 5 in the Appendix shows our complete MCA Questionnaire. Four questions are not listed in Table 5: a generic permission question, a generic intent filter question, the "contains native code" question, and the "contains zip files" question. A generic question simply groups unrelated attributes that we empirically found to be less important. To avoid the additional "noise" resulting from creating an additional question for each attribute, we grouped them into a single disjunctive question. Therefore, applications having one of these attributes will have something in common. The generic permission and intent filter questions simply contain all of the remaining permissions and action strings, respectively.

4.3 Poll Selection and Quantification

MAST runs MCA on multiple polls to create rough indicators of malicious behavior. Each poll asks the MCA questionnaire to a specific subset of applications. Polls fill two purposes: 1) they group related applications together so that uniqueness within the group is more specific, and 2) they allow MAST to select which characteristics are most effective in identifying current malware trends.

The polls used by MAST should reflect current malware trends, and therefore could change as malware evolves. Furthermore, poll selection characteristics are not necessarily the same as the characteristics used for MCA questions. That said, all of our MCA questions are potential poll characteristics. However, we select only those polls that reflect a certain inclination to malicious behavior. The MAST polls selected by our training process described in Section 5.3 are shown in Table 2.

Note that the application subsets defined by polls are intentionally not disjoint—an application that has both SMS and PACKAGE

[2]Dynamic broadcast receivers that define intent filters at runtime are excluded as they require costly program analysis to retrieve.

Table 2: Characteristics used to define polls. Each poll defines a subset of applications directed to MCA.

Characteristic	Possible Malicious Behavior
SMS perms	SMS trojans, SMS spam
PACKAGE perms	Installing malicious apps, Uninstalling antivirus apps
BOOT intent-filter (default)	Spyware apps that want to autorun on startup
BOOT intent-filter (priority)	Spyware apps that want to autorun before anti-malware apps are started
Native code	Native exploits (rage-against-the-cage)
Zip files	Zipped payload containing malicious apps
COMM. intent-filter (default)	Applications that use incoming calls or SMS as activation triggers
COMM. intent-filter (priority)	Applications that hide incoming calls or SMS from users

permissions will be analyzed in two polls. When MAST merges the MCA results for the selected polls, an application that exhibits multiple malware-requisite characteristics will thus stand out further in the MAST ranking.

Finally, MAST quantifies the MCA results for each poll. Section 3 demonstrated how MCA can be used as a visualization tool. However, MCA can also be used to quantify the deviation of an application from the norm. We quantify a ranking for each MCA by calculating the χ^2 distance of each application from the barycenter (point of commonality) of all apps in the analysis. This quantification provides a *poll score* for each analyzed application.

4.4 Merging Poll Results

MAST merges the individual poll scores to create an accurate MAST ranking. The merge must ensure: 1) results for one poll do not overshadow another; 2) the number of apps in a poll is considered; and 3) the number of polls an application participates in influences the total ordering. We now describe this merge process.

Normalization: Once each poll is ranked, MAST normalizes the poll scores by scaling each poll i from domain $[min_i, max_i]$ to domain $[0, 1]$, where $min_i \geq 0$.

Poll Size Scaling: To account for the number of applications in a poll, MAST scales the normalized poll scores such that the smaller the set of apps in a poll, the larger the contribution of the outlier applications. For each poll, MAST scales the normalized poll score using the subset of applications $A_i \subseteq A$ present in the poll:

$$[0, 1] \rightarrow [\left(1 - \frac{|A_i|}{|A|}\right), 1]$$

Combining and Sorting Results: Finally, to determine the overall ranking, MAST calculates a *MAST score* for each application by summing its normalized and scaled poll scores. The greater the number of subsets an application appears in, the larger its MAST score, which reflects a higher degree of relative suspicion. Applications are then sorted by MAST score, producing the *MAST ranking*.

4.5 MAST Ranking

The final output of MAST is a list of applications ranked in order of their dissimilarity to the population of other applications. Due to poll selection, this list indicates a relative degree of suspicion. Applications ranked higher in the list are more likely to be malicious than those appearing at the end. Thus, when performing triage, investigation should start with the highest MAST ranking.

Note that it may then make sense to classify apps as being in a "percentile of suspicion;" that is, if an app is within the top 1% of apps in the MAST ranking, it could be said that that application is within the top 1% of suspicious apps. An app can be defined as suspicious if it exhibits behaviors common to malicious apps. We caution that MAST is not Bayesian — MAST does not specify or even imply the actual likelihood of maliciousness of any application, even given a percentile of suspicion. It is certainly *not* the case that an application in the first percentile of suspicion (1%) is 99% likely to be malicious.

4.6 MAST Implementation

Our implementation of MAST consists of two main steps: obtaining metainformation from applications and processing that metainformation. A Python program unzips every application package, checks all files for zip archives and native code, decompresses the `AndroidManifest.xml` file, computes the questionnaire results, and writes that information to a "MAST table". Table computation is the major bottleneck, but is fortunately trivially parallelizable (though our implementation processes apps serially). Once the questionnaire is complete, the table is read by a program written in R, which is a popular language for statistical analysis that also has an MCA library. The R program parses the MAST table, selects apps based on polls, runs an MCA of every poll, quantifies the poll results, then merges them to produce a final MAST ranking.

5. TRAINING MAST

MAST merges the MCA poll results that roughly indicate maliciousness to magnify the malicious characteristics of malware. These polls are directed at applications that exhibit specific characteristics found in malware (but also benign applications). Choosing which polls MAST should use requires careful consideration. From a high level, we want to choose polls that cover the characteristics of known malware, but we do not want to over-train MAST for any specific malware type. At the same time, we want to select polls for characteristics that are more common in malware than benign applications. In this section, we describe our process of selecting polls for MAST (previously discussed in Table 2).

5.1 Training Data

In order to train MAST, we created a simulated market. This market consists of 14,888 popular free applications from Google Play, 141 samples of known malware from the Contagio mobile malware repository [43] as of October 2011 and 591 malware samples found by Zhou et al. [53]. Here, we assume the Android Market applications are mostly benign, but keep in mind the potential for malicious and questionable applications when training MAST.

Google Android Market: Properly training MAST requires a very large set of benign applications. Manually downloading these applications on a phone is not an option, so we developed a standalone snapshot tool using the unofficial Android Market API [2]. Our tool downloaded the majority of the top 500 free applications within each of the 34 market categories. By selecting most popular applications, we maximize our chances of selecting benign applications. Our dataset, taken on January 20, 2012, included applications for the "T-Mobile" carrier, the "passion" device (Nexus One), and Android API level 8 (Froyo) and below. Due to failed downloads and categories that had fewer applications, our snapshot contains a total of 14,888 applications. This dataset represents approximately 2.1% of the estimated over 700,000 applications in Google Play [52]. Results of real-world triage in Section 6 show that even 2.1% of the market is sufficient to accurately train MAST.

Combined malware training set: Our combined malware set of 732 applications includes malware samples that steal personal data

and receive commands from an attacker's command and control server, send SMS to premium numbers, place calls to premium numbers, and/or otherwise spy on the user (including tracking the user's location). Some malware samples display more than one of these malicious behaviors. Android malware authors often embed the same exploit in multiple applications, which leads to the existence of malware families. In addition to malware, the malware set contains several examples of grayware. Two grayware examples include applications designed to spy on the SMS or GPS activity of one's spouse without his/her consent, and applications designed to gain root access for the user. Because there are no guarantees that these samples do not abuse their abilities for malicious purposes, and because MAST is focused on triage for potentially malicious behavior, we include the grayware in our training data set. Table 6 in the appendix presents the composition of the 732 samples in our combined malware set.

5.2 Evaluation Metrics

MAST ranks a given set of applications according to their relative suspiciousness. This ordering indicates where malware researchers should first allocate their resources. To measure the effectiveness of MAST, we use receiver operating characteristic (ROC) curves, which provide a plot of true positive (TP) and false positive (FP) rates with regard to a threshold parameter. In the case of MAST, the threshold parameter is the top percentage of apps in the MAST ranking that are scanned. The positives in the ROC curve are the apps that are chosen to be scanned, with the true positive being the malicious ones and false positives being the non-malicious ones.

When evaluating the performance of MAST, we compare three triage techniques.

- *No Triage*: When no intelligent approach can be applied, triage is equivalent to random guessing. We model the "no triage" scenario by assuming the malicious applications are uniformly distributed within a random ordering of the applications. The corresponding ROC curve is the line of no discrimination and has a slope of one.
- *Kirin-based Triage*: Kirin [16] defines possible malicious behaviors using nine permission-based conjunctive rules. Two rules are concerned with detecting SMS malware. Used as a triage technique, malware researchers apply Kirin to create two sets of applications: high priority and low priority. The high priority set consists of applications that fail any of the Kirin rules. We model Kirin-based triage by assuming the malware binned in each set is uniformly distributed in a random order. The corresponding ROC curve consists of two slopes, indicating the division between the high and low priority sets.
- *MAST-based Triage*: MAST provides an ordering for triage. We use the MCA questionnaire defined in Section 4.

5.3 Selecting Polls

As discussed in Section 4.3, a poll is essentially a subset of applications that have similar functionality. The purpose of MAST training is to select a combination of polls that individually act as rough indicators of suspicious behavior and that improve the overall accuracy of the MAST ranking. As MAST questions are directed towards functionality, each question and answer pair defines a MAST poll. However, we narrowed down the set of possible polls to those that contained at least 10% of the malware in our training set. On the 15 polls that showed a weak correspondence to malware, we ran a brute-force analysis to determine the optimal

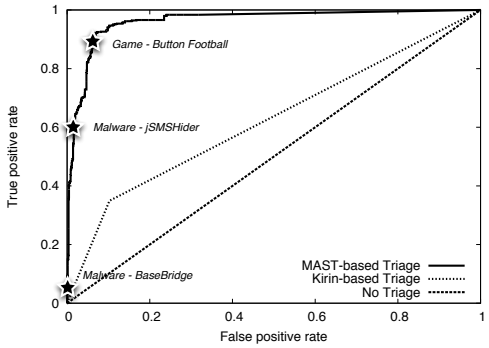


Figure 7: Training data ROC curve (732 malapps in 15,620 apps)

combination of polls. Note that we did not need to run MAST completely for each experiment, but only the merging of the individual poll results. Table 2 presents our final poll selection.

An interesting observation from our poll selection process is that all the polls selected in our optimal combination show good individual poll performance. Figure 3a shows that the SMS permission poll alone ranks nearly 70% of the malware samples from our combined malware training dataset with a false positive rate of 5%. The package permission poll only includes approximately 30% of the malware; however, it ranks that malware with a false positive rate of 0.8%. However, as all polls with good individual performance do not improve overall MAST performance, we can only use this observation to further reduce the training search space of future experiments by discarding polls with poor individual performance (Figure 3c, 3d). Discarding polls with poor performance does not affect the ability of MAST to detect new kinds of malware, as long as we use a sufficient number of polls that are indicative of malicious functionality. We validate the optimality of our poll count by observing that MAST results tend to peak at poll sets having sizes between 8 and 10 and drop as the number of polls is decreased or increased.

5.4 Combining Polls

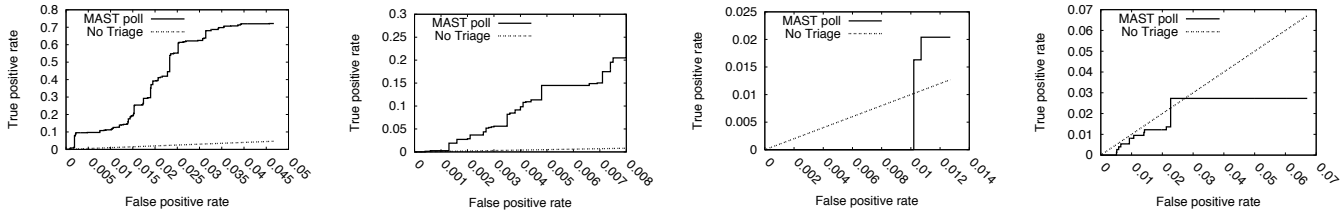
We previously claimed that: a) MCA classifies malware as outliers and, b) combining multiple polls reduces analysis effort. We now prove these claims by tracking three applications as they are processed by MAST: two malware samples (BaseBridge and jSMSHider) and one randomly selected app, Button Football (a soccer game app), that appeared in two polls. In these plots, the tracked applications are highlighted with a black star.

Figure 4 tracks the BaseBridge malware sample in four polls: SMS permissions, BOOT intent-filter (priority) poll, zip file poll, and COMMUNICATION intent-filter (priority) poll. The crosshair in the plot denotes the barycenter (i.e., point of commonality). The BaseBridge sample is a clear outlier in each of the SMS, BOOT-priority, zip file, and COMMUNICATION-priority polls. Being a distinct outlier, the BaseBridge application is ranked 0.25%.

Figure 5 tracks the jSMSHider malware sample in two polls: PACKAGE permissions and zip file filter. The sample is outside the main cluster in the PACKAGE permission poll, but not far from the barycenter in the zip file poll. In this case, combining the two polls has a clear advantage for ranking the malware sample.

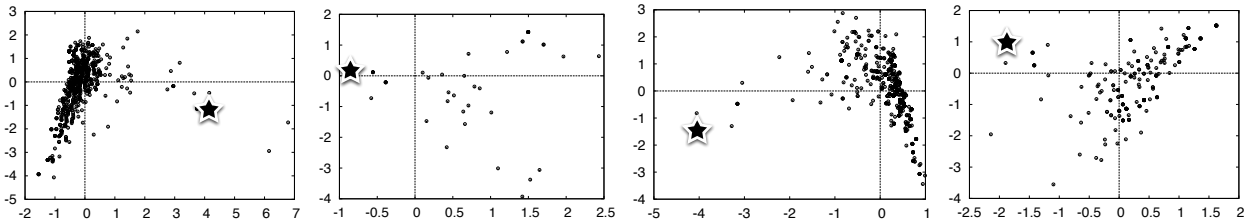
Figure 6 tracks Button Football, a soccer game application in the two individual polls matching its characteristics: BOOT intent poll and Native code poll. Here, Button Football is grouped in the main clusters, contributing to its relatively lower MAST rank.

Finally, Figure 7 shows the ROC curve after combining our selected polls. MAST ranks BaseBridge and jSMSHider malware apps very high (0.25% and 4.4%). These results demonstrate how



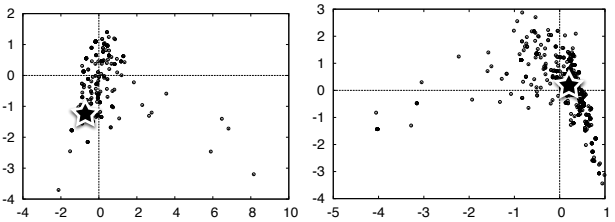
(a) SMS Permission Poll (b) PACKAGE Permission Poll (c) CALENDAR Perm. Poll (d) ACCOUNT Permission Poll

Figure 3: Isolated Poll Performance. CALENDAR and ACCOUNT were not selected due to poor performance.



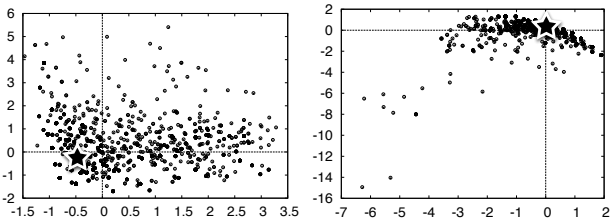
(a) SMS Permission Poll (b) BOOT Intent (Priorty) Poll (c) Zip File Poll (d) COMM. Int. (Priority) Poll

Figure 4: Tracking BaseBridge malware (MAST rank: 0.25%)



(a) PACKAGE Permission Poll (b) Zip File Poll

Figure 5: Tracking jSMSHider malapp (MAST rank: 4.4%)



(a) Boot Intent (Default) Poll (b) Native Code Poll

Figure 6: Tracking Button Football (MAST rank: 9.7%)

the multiple polls aid in giving malware higher rankings. Button Football has a lower ranking (9.7%), but is still within the top 10% of applications to receive analysis. However, we note that Button Football's inclusion in the BOOT intent-filter default poll is interesting for a game application. In the spirit of diagnosis after triage, we analyzed the Button Football app to find that it uses AirPush, an aggressive notification ad library. AirPush goes beyond in-app ads by generating notifications that persuade users to download apps. Further, it starts itself at boot, so the user can see ads even when he is not using any application. The higher MAST ranking of Button Soccer highlights that, from the point of view of MAST, the distinction between undesirable and malicious behavior is thin.

Comparing the ROC curves for MAST-based triage, Kirin-based triage, and no triage, the advantage of MAST is clear. MAST gathers 90% of the malware samples while suffering a false positive rate of just 6.5%. Furthermore, it collects 95% of the malware samples at a false positive rate of just 11.2%. This result merely verifies that our training process is sound. Section 6 provides a real-world evaluation of MAST.

6. REAL MARKET TRIAGE

We now measure the effectiveness of MAST on real third-party markets, *which were not used for training.*

6.1 Experimental Setup

We selected three application markets to evaluate MAST: Ndoo, Anzhi, and Softandroid. All three make their apps available free of charge online, therefore we were able to download all apps in each market. *All markets were found to host some amount of malware.*

To validate MAST's ability to highly rank malware, we need a perception of which apps in a market are malicious. We stress that this knowledge is not required for MAST to rank the applications in the market. We use two tools to identify malicious apps: Androguard and Virus Total. Androguard is a well-known, open-source project with volunteer-submitted definitions and provides a lightweight signature-based malware detection tool. Our analysis is based on the signature definitions from February 5, 2012. Virus Total is an online service that scans submitted files with tens (often more than 40) of up-to-date commercial antivirus products and provides the results from each AV product. Our markets were last scanned by Virus Total on February 11, 2012.

Androguard has a low true positive rate as its malware signatures are updated very slowly. On the other hand, some AV products used in Virus Total have a high false positive rate. To overcome these limitations, we use a hybrid approach that tags applications as malware if either Androguard or at least three out of the "Top 5" AV products in Virus Total tag the application as malware. The "Top 5" AV products (GData, Avast, Kaspersky, BitDefender, and FSecure) were chosen based on their accuracy in the detection of known malware samples. The distribution of the malware we discovered in these markets is presented in Table 6 in the appendix.

Ndoo: The Ndoo market is a Chinese app market. On October 25, 2011, the market contained 4,324 apps. Of these apps, 26 were considered malicious by Androguard or Virus Total. At the time of data collection, Ndoo claimed that its "comprehensive software testing & certification procedures contribute to the reputation of [the] developer." No further details were provided on its English website about its security evaluation procedures.

Anzhi: Like Ndoo, Anzhi is a market catering to Chinese Android users. Anzhi contains far more apps though: on January 31, 2012,

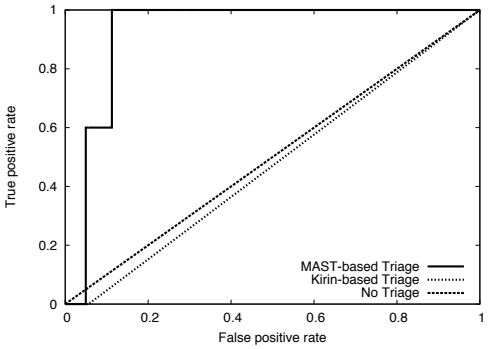


Figure 8: Softandroid ROC curve (5 malapps in 3,626 apps)

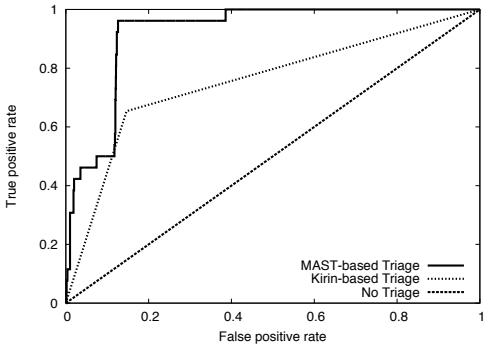


Figure 9: Ndoo ROC curve (26 malapps in 4,324 apps)

the market contained 28,760 apps. 166 of these were considered malicious by Androguard or Virus Total (six of these were later found to be false positives).

Softandroid: Softandroid is a Russian application market hosting 3,626 apps on February 7, 2012. VirusTotal or Androguard marked six of these as malicious (one was later found to be a false positive).

6.2 MAST Results

Figures 8, 9, and 10 show ROC curves for the Softandroid, Ndoo, and Anzhi markets, respectively.

As the SoftAndroid market contained only five instances of malware, its ROC curve is the easiest to analyze. Three of the apps marked as malicious are within the top 5% of the MAST rankings. The remaining two apps are within the top 11.2% of the MAST rankings. For this market, MAST is able to highly rank all malicious applications, even when no malicious applications violate a Kirin rule. This validates the choice of disjunctive polls.

The Ndoo market contained 26 malicious apps. MAST ranks 25 of these – 96.1% – in the top 12.9% of the MAST rankings. Half of these malicious apps are ranked in the top 7.5%, and the highest-ranked malicious app is the 8th app in the rankings. While Kirin ranks a majority of the malicious apps in this market, MAST catches more and orders them earlier than Kirin.

The Anzhi market contained 160 malicious apps. 60% of these were ranked in the first 2.1% of the market. 85% of the malicious apps are located in the top 10% of the MAST rankings, and 95.5% of the malicious apps are located in the top 25% of the rankings. As with Softandroid, Kirin rules would have failed to rank the majority of malicious apps in the Anzhi market.

We compare MAST's ROC curves to those of Peng et al.'s best technique, HMNB [44], using their area under the curve (AUC) figures. The AUC of a ROC curve gives an overall indication of quality of the curve. Peng et al. [44] cite an AUC of 0.9281 when testing against market data that does not overlap with their testing data, while the AUC for Anzhi, SoftAndroid, and Ndoo are 0.9362,

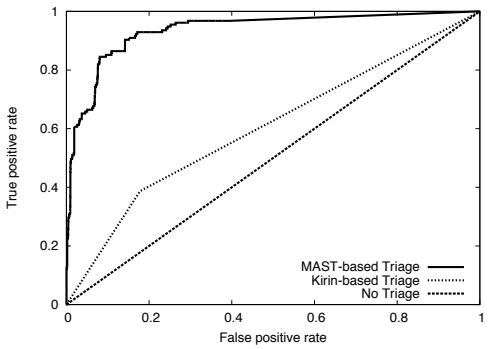


Figure 10: Anzhi ROC curve (160 malapps in 28,760 apps)

0.9252, and 0.9207 respectively. (None of these markets have overlapping apps, according to the SHA-256 of their packages). These results show that MAST provides comparable or better malware triage than HMNB would. We emphasize that Peng et al. [44] test on two snapshots of the Android market with malware from an external dataset added, while we test on real markets with malware actually present in the market.

6.3 Performance Analysis

Because the purpose of a triage is to act as a fast first step to optimize a later heavy analysis, we compare MAST and other analysis tools. All tests were run in a VirtualBox VM configured with a single CPU and 2GB of memory with a 10GB virtual hard disk running Ubuntu Linux 10.04 (Windows XP SP3 in the case of the TrendMicro test). Market files were stored on the host's 16TB 12-drive RAID 6 array and accessed using VirtualBox shared folders[3]. The host machine has 2 Intel E5620 Xeon quad-core processors and 48 GB RAM and runs Debian Squeeze.

Table 3 provides a comparison of the time to run MAST compared to heavier, more comprehensive methods. We note that MAST is meant to complement these methods, not replace them. The comparison is made only to prove that MAST is light-weight enough that the cost of using MAST is amortized by more efficient analysis of apps. First, MAST performance numbers are provided for two markets: Ndoo and Android. The runtime of MAST is overwhelmingly bound by collecting attributes from each app; performing the MCA for all polls requires less than 1% of the total time presented in the table. MAST is comparable in time to Baksmali [5], a popular open-source disassembler for Dalvik bytecode. Disassembly is a first minimum step for manual analysis, so it represents a reasonable lower bound on manual analysis. Actual manual analysis would of course take minutes or hours per application. MAST is 4.3 times faster than the command line version of TrendMicro Antivirus [50] used to perform signature detection over the Ndoo market. On a selection of 23 randomly selected apps from the Google market, ded [15] (a decompiler for Android applications) runs four orders of magnitude slower than MAST.

For compute-intensive analyses of markets, MAST increases the likelihood of identifying malicious apps in a reasonable amount of time. Moreover *the cost of MAST is so low compared to techniques such as ded that the overhead of running MAST is negligible.*

6.4 Post-triage Analysis

After having successfully performed triage, we made two interesting observations by performing deeper analysis across suspicious applications from each of the markets.

[3]Virtual folders were found by microbenchmarks to be faster than accessing from the virtual hard disk.

Table 3: Analysis Technique Timing Comparison

Tool	Input	Total Time	Time/App
Baksmali	Ndoo Market	24.63m	0.34s
MAST	Ndoo Market	32.18m	0.45s
MAST	Android Market	111.5m	0.45s
TrendMicro AV	Ndoo Market	140.7m	1.95s
ded	23 Random Apps	487.7m	1272.29s

Table 4: Rampant use of the default key from the Android codebase to sign applications across markets

Market	Number of applications signed using the default key
Google Play	11
Contagio + Malware	156
Softandroid	245
Ndoo	510
Anzhi	750

False positives in commercial antivirus tools: After the initial evaluation of MAST, we found five applications in our malware training set, six in Anzhi, and one in SoftAndroid that were classified as SMS trojans (HippoSMS and RogueSPPush) by antivirus tools, yet were ranked poorly by MAST. Upon inspection, none actually had SMS permissions, meaning that it could not exercise any malicious code that might be present. In effect, these are false positives.[4]. To verify this, we selected one of these apps (classified as RogueSPPush) for further manual analysis. We found that this app did have the malicious RogueSPPush code, but was effectively "dead" as it did not declare the SMS intent filter that executes that code. We did not find evidence of a root exploit or any other method to violate Android's permission system. This finding highlights that just as MAST highly ranks apps that could exhibit malicious behavior, it ranks apps without this explicit ability lower.

Default Android key used to sign applications: Android requires all applications to be signed to ensure that an application can be upgraded only by the developer who created it. Thus, there is no need for a certificate authority or PKI — self-signed certificates suffice, as long as the security of the private key is maintained by the developer. However, we found that 1,672 applications across the markets (distribution in Table 4) used the default key present in the Android codebase to sign their applications.

In terms of security, using the default private key is equivalent to posting your private key in a public forum. Any malicious author can update the applications signed with the default key and replace them with malicious code. The fact that we found these applications in the official Google market indicates the need to check for use of the default key when applications are submitted to the market. Given the use of this key in applications in the alternative markets, we highly recommend that the package installer in production Android phones blacklists the default key.

We also found a significant number of applications across markets signed with the same private key as malicious applications. However, we leave further analysis of those results as future work.

6.5 MAST ranking limitations

Triage in principle is a "best effort approach". With MAST, we aim to increase the likelihood of finding malware in the lowest percentiles of suspicion, but our results can only be as good as the polls we conduct. Our current efforts attempt to select classes of attributes (e.g., permissions, intents, etc.) that are absolutely necessary for malware attempting to perform a specific task to declare (e.g., SMS trojans must ask for SMS permissions, or their malicious code simply will not run). However, if new classes of malware attempt to abuse other protected or unregulated [36] interfaces, MAST is unlikely to rank them highly. This is analogous to a medical triage case where a patient has no symptoms of illness, but is infected with an unknown disease. These scenarios can be easily addressed in MAST as soon as new "zero-day classes" are discovered. Specifically, new polls can be added to the infrastructure, and new malicious applications can just as quickly be flagged.

[4]We exclude these apps from malware counts throughout the paper.

7. CONCLUSION

Application markets simplify the distribution of consumer software. The benefits have not been lost on malware authors: application markets are the primary means of distributing smartphone malware. Preventing malware in markets is extremely difficult. Market maintainers simply do not have the computational or personnel resources to thoroughly or deeply inspect the large number of applications submitted each day. To address this challenge, we propose MAST. The goal of MAST is to direct available analysis resources to the most suspicious applications. To do this, MAST uses Multiple Correspondence Analysis (MCA) to measure the correlation between declared indicators of functionality required to be present in application packages. We described how to parameterize MAST using current malware trends and then demonstrated its value by using it to successfully perform triage on three third-party markets.

The concepts underlying MAST transcend malware discovery in two ways. First, we show that MCA, a tool primarily used in the social sciences, has tremendous potential for security, specifically when the adversary must declare (i.e., commit to) functional specifications. The key contribution of MCA is its ability to establish relationships between otherwise incomparable information. Second, we believe that security triage tools that quantify perception are essential to protect systems from intelligent adversaries. Triage is neither diagnosis nor treatment. Rather, such tools make security analysis more methodological, and less reliant on a "gut feeling."

8. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of VirusTotal for this work. This work was supported in part by the US National Science Foundation under grant numbers DGE-1148903, CNS-0916047, CNS-0952959, and TWC-1222699. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] H. Abdi and D. Valentin. Multiple correspondence analysis. In *Encyclopedia of Measurement and Statistics*, page 13. Sage, California, 2007.
[2] Android market API. http://code.google.com/p/android-market-api/.
[3] Anzhi Market. http://www.anzhi.com.
[4] Apple app store, 2012. http://www.apple.com/iphone/from-the-app-store/.
[5] Baksmali, 2012. http://code.google.com/p/smali/.
[6] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, page 73. ACM Press, 2010.
[7] C. Beaumont. Apple iPhone 'kill switch' discovered, August 2008. http://www.telegraph.co.uk/technology/3358115/Apple-iPhone-kill-switch-discovered.html.
[8] Blackberry app world, 2012. http://appworld.blackberry.com/webstore/.
[9] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, page 225. ACM Press, 2008.
[10] T. Bray. Exercising Our Remote Application Removal Feature, June 2010. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.
[11] J. Burns. Developing Secure Mobile Applications for Android. iSEC Partners, Oct. 2008. http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.
[13] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the ISOC Network & Distributed System Security Symposium (NDSS)*, 2011.
[14] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.
[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, 2011.
[16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, page 235. ACM Press, 2009.
[17] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security & Privacy Magazine*, 7(1):50–57, January/February 2009.
[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, Chicago, Illinois, USA, Oct. 2011.
[19] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *ACM Workshop on Security and Privacy in Mobile Devices*, Chicago, Illinois, USA, Oct. 2011.
[20] GFan Market. `http://www.gfan.com/`.
[21] Google play, 2012. `https://play.google.com/store/apps`.
[22] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2012.
[23] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.
[24] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.
[25] X. Jiang. Questionable Android Apps – SndApps – Found and Removed from Official Android Market, July 2011. `http://www.csc.ncsu.edu/faculty/jiang/SndApps/`.
[26] X. Jiang. Security Alert: New Android SMS Trojan – YZHCSMS – Found in Official Android Market and Alternative Markets, June 2011. `http://www.csc.ncsu.edu/faculty/jiang/YZHCSMS/`.
[27] X. Jiang. Security Alert: New Stealthy Android Spyware – Plankton – Found in Official Android Market, June 2011. `http://www.csc.ncsu.edu/faculty/jiang/Plankton/`.
[28] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, page 239. ACM Press, 2008.
[29] A. Kingsley-Hughes. So that's what happens when you highlight an iOS security hole, November 2011. `http://www.zdnet.com/blog/hardware/so-thats-what-happens-when-you-highlight-an-ios-security-hole/16078`.
[30] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, volume 5758, pages 244–264, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
[31] H. Lockheimer. Android and Security. Google Mobile Blog, Feb. 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.
[32] Lookout Mobile Security. Mobile threat report. Technical report, Lookout Mobile Security, Aug. 2011.
[33] Lookout mobile security, 2012. `https://www.mylookout.com/`.
[34] J. Lowensohn. iPhone lock-screen password app pulled, June 2011. `http://news.cnet.com/8301-27076_3-20071405-248/iphone-lock-screen-password-app-pulled/`.
[35] K. Mahaffey. Security Alert: DroidDream Malware Found in Official Android Market, March 2011. `http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/`.
[36] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (sp)iPhone: Decoding Vibrations From Nearby Keyboards Using Mobile Phone Accelerometers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.
[37] P. McDaniel and W. Enck. Not so great expectations: Why application markets haven't failed security. *IEEE Security & Privacy*, 8(5):76–78, Oct. 2010.
[38] Min Zheng, Patrick P.C. Lee, and John C.S. Lui. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'12)*, Heraklion, Crete, Greece, July 2012.
[39] Ndoo market. http://www.nduoa.com/.
[40] NetQin Mobile Security, 2012. `http://www.netqin.com/en/`.
[41] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. Forecast: skimming off the malware cream. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 11–20, New York, NY, USA, 2011. ACM.
[42] Nicholas J. Percoco and Sean Schulte. Adventures in BouncerLand. In *Blackhat USA*, Las Vegas, NV, 2012.
[43] M. Parkour. Contagio mobile malware MiniDump. http://contagiominidump.blogspot.com/.
[44] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, page 241–252, New York, NY, USA, 2012. ACM.
[45] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 301–310, Washington, DC, USA, 2008. IEEE Computer Society.
[46] B. L. Roux and H. Rouanet. *Multiple Correspondence Analysis*. Number 163 in Quantitative Applications in the Social Sciences. SAGE Publications, Los Angeles, California, USA, 2010.
[47] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, page 13–22, New York, NY, USA, 2012. ACM.
[48] R. E. Schapire. The Boosting Approach to Machine Learning: An Overview. In *Nonlinear Estimation and Classification*. Springer, 2003.
[49] SoftAndroid Market. `http://softandroid.ru`.
[50] Trend Micro Command Line Antivirus Scanner, 2012. `http://esupport.trendmicro.com/solution/en-us/0117058.aspx`.
[51] Windows Phone: Marketplace, 2011. `http://www.windowsphone.com/en-US/marketplace`.
[52] B. Womack. Google says 700,000 applications available for android, Oct. 2012.
[53] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*, 2012.
[54] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2012.
[55] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *TRUST*, pages 93–107, 2011.

APPENDIX

A. FORMAL DESCRIPTION OF MCA

MCA has been independently discovered numerous times [1]; accordingly, descriptions of the methods and the terms used differ from author to author. We follow the description of Abdi et al. [1], with terminology and insights borrowed from La Roux et al. [46].

MCA constructs a cloud of points representing individuals by encoding the data as a matrix with a column for each possible *answer*, not question. Each of these columns is termed a "category". Let K be the set of categories. The contents of an individual element of this matrix δ_{ik} for $i \in I$, $k \in K$ will be a "1" for an individual choosing a category, and "0" otherwise. This matrix is called an *indicator matrix* and is represented as $\mathbf{X}$. This indicator matrix will be of dimensions $I \times K$, and is defined such that each row sum is constant:

$$\sum_{k=1}^{K} \delta_{ik} = N$$

for all $i \in I$. The restaurant example would have two categories describing "Attire" ("Formal" and "Casual") and three categories for "Cost" ("High", "Med", and "Low").

Once an indicator matrix of the data is constructed, MCA computes a *probability matrix* $\mathbf{Z} = |I|^{-1}\mathbf{X}$ and a supplemental probability matrix $\mathbf{Z_0} = \mathbf{Z} - \mathbf{rc^T}$, where $\mathbf{r}$ and $\mathbf{c}$ are column vectors with $\mathbf{r_i} = \sum_{k=1}^{K} \delta_{ik}$ for all i in I and $\mathbf{c_k} = \sum_{i=1}^{I} \delta_{ik}$ for all k in K. Essentially $\mathbf{r}$ and $\mathbf{c}$ are the respective row and column sum vectors of $\mathbf{Z}$. Then MCA constructs the matrix $\mathbf{H} = \mathbf{D_r^{-\frac{1}{2}}} \mathbf{Z_0} \mathbf{D_c^{-\frac{1}{2}}}$ where $\mathbf{D_r} = diag(\mathbf{r})$ and $\mathbf{D_c} = diag(\mathbf{c})$.

To arrive at the new coordinates for the individuals represented by the rows of the indicator matrix, MCA computes the singular

Table 5: MCA Questionnaire* composed of disjunctive questions.

Permission Question	**Included permissions** (All permissions have the "android.permission" prefix.)
ACCOUNT	ACCOUNT_MANAGER, AUTHENTICATE_ACCOUNTS, GET_ACCOUNTS, MANAGE_ACCOUNTS, USE_CREDENTIALS
AUDIO	RECORD_AUDIO, MODIFY_AUDIO_SETTINGS
BOOKMARKS	WRITE_HISTORY_BOOKMARKS, READ_HISTORY_BOOKMARKS
CALENDAR	WRITE_CALENDAR, READ_CALENDAR
CONTACTS	WRITE_CONTACTS, READ_CONTACTS
FILESYSTEM	MOUNT_FORMAT_FILESYSTEMS, MOUNT_UNMOUNT_FILESYSTEMS
GENERIC_SETTINGS	WRITE_SECURE_SETTINGS, WRITE_SETTINGS
INTERNET	INTERNET
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, ACCESS_LOCATION_EXTRA_COMMANDS, ACCESS_MOCK_LOCATION, CONTROL_LOCATION_UPDATES, INSTALL_LOCATION_PROVIDER
NETWORK	ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, BLUETOOTH, BLUETOOTH_ADMIN, BROADCAST_WAP_PUSH, CHANGE_NETWORK_STATE, CHANGE_WIFI_MULTICAST_STATE, CHANGE_WIFI_STATE, NFC, RECEIVE_WAP_PUSH, WRITE_APN_SETTINGS
PACKAGE	DELETE_PACKAGES, INSTALL_PACKAGES, BROADCAST_PACKAGE_REMOVED, GET_PACKAGE_SIZE
PHONE_STATE	MODIFY_PHONE_STATE, READ_PHONE_STATE
CALLS	CALL_PHONE, CALL_PRIVILEGED, PROCESS_OUTGOING_CALLS, USE_SIP
SMS	READ_SMS, SEND_SMS,WRITE_SMS, RECEIVE_SMS, BROADCAST_SMS, RECEIVE_MMS
Intent Filter Question	**Included action strings from intent filters** (Unless otherwise indicated, all actions strings have the "android.intent.action" prefix.)
BOOT	BOOT_COMPLETED, REBOOT
COMMUNICATIONS	ANSWER, CALL, CALL_BUTTON, DIAL, android.provider.Telephony.SMS_RECEIVED, PHONE_STATE
MEDIA	MEDIA_BAD_REMOVAL, MEDIA_BUTTON, MEDIA_CHECKING, MEDIA_EJECT, MEDIA_MOUNTED, MEDIA_NOFS, MEDIA_REMOVED, MEDIA_SCANNER_FINISHED, MEDIA_SCANNER_SCAN_FILE, MEDIA_SCANNER_STARTED, MEDIA_SHARED, MEDIA_UNMOUNTABLE, MEDIA_UNMOUNTED
PACKAGE	MANAGE_PACKAGE_STORAGE, MY_PACKAGE_REPLACED, NEW_OUTGOING_CALL, PACKAGE_ADDED, PACKAGE_CHANGED, PACKAGE_DATA_CLEARED, PACKAGE_FIRST_LAUNCH, PACKAGE_INSTALL, PACKAGE_REMOVED, PACKAGE_REPLACED, PACKAGE_RESTARTED, MANAGE_PACKAGE_STORAGE, MY_PACKAGE_REPLACED, NEW_OUTGOING_CALL, PACKAGE_ADDED, PACKAGE_CHANGED, PACKAGE_DATA_CLEARED, PACKAGE_FIRST_LAUNCH, PACKAGE_INSTALL, PACKAGE_REMOVED, PACKAGE_REPLACED, PACKAGE_RESTARTED
POWER	BATTERY_CHANGED, BATTERY_LOW, BATTERY_OKAY, ACTION_POWER_CONNECTED, ACTION_POWER_DISCONNECTED, POWER_USAGE_SUMMARY, ACTION_SHUTDOWN
WALLPAPER	WALLPAPER_CHANGED, SET_WALLPAPER

* Discussed in Section 4.2, we additionally include a generic permission question, a generic intent filter question, a native code question, and a zip file question

Table 6: Malware distribution across markets

Malware Family	Malware Description	Training Set	Anzhi	Ndoo	Softandroid
ADRD	Information Stealer	28		1	
Anserver Bot	Downloads malicious payloads, Information Stealer	5			
Asroot	Native root exploit	8			
BaseBridge	Root exploit, SMS, CALL Trojan	116	19		
Bgserv	Information Stealer, SMS Trojan	7		1	
Boxer	SMS Trojan				2
DroidDream	Root Exploit, Information Stealer	20			
DroidDreamLight	Information Stealer	23	18		
DroidKungFu	Root exploit, Downloads malicious payloads	179	53	19	
Geinimi	SMS Trojan, Information Stealer	72			3
GoldDream	SMS, CALL spy, Bot capabilities	27	12		
GPSSMSSpy	Location and SMS spy	6			
HippoSMS	SMS Trojan	5			
jSMSHider	SMS Trojan targetting custom ROMs	18			
KMIN	Information Stealer	40			
NickySpy	SMS, GPS, CALL spy	3			
Pjapps	Information Stealer, SMS Trojan	52			
Plankton	Downloads malicious payloads, Information Stealer	10	3		
RogueSPPush	Automatically subscribes to premium SMS services	3			
SndApps	Information Stealer	10			
YZHC	SMS Trojan	34	1		
zHash	Native root exploit	12			
Zsone	SMS Trojan	12	1	1	
Other	-	42	53	4	
Total	-	732	160	26	5

value decomposition of $\mathbf{H}$: $\mathbf{H} = \mathbf{P\Delta Q^T}$. The principal coordinates of each individual and category are described by the matrices

$$\mathbf{Y_I} = \mathbf{D}_r^{-\frac{1}{2}}\mathbf{P\Delta} \; and \; \mathbf{Y_K} = \mathbf{D}_c^{-\frac{1}{2}}\mathbf{Q\Delta}$$

From this, the χ^2 distance of an individual from the barycenter of all individuals is computed as $\mathbf{d} = diag(\mathbf{Y_I Y_I^T})$

Eigenvalues are also computed from the singular value decomposition of $\mathbf{H}$. The eigenvalues, termed *inertia* in MCA, are the variances of the principal axes, and are defined as $\mathbf{\Lambda} = \mathbf{\Delta}^2$. These variances sum to the total variance of the cloud formed by the indicator matrix: $Var(\mathbf{X}) = \mathbf{e_i} diag(\mathbf{\Lambda})$ where $\mathbf{e_i}$ is the column vector of ones of appropriate size. The concept of inertia is important to the interpretation of an MCA analysis because it describes which principal axes are relevant. In most cases, the relevant principal axes are determined as those with inertia greater than $\frac{Var(\mathbf{X})}{l}$ where l is the total number of eigenvalues. The effectiveness of MCA is predicated on the fact that the number of principal axes with sufficient inertia is quite small (2 or 3) even with large N.