Android Malware Development on Public Malware Scanning Platforms: A Large-scale Data-driven Study

Heqing Huang[1,2], Cong Zheng[3], Junyuan Zeng[4], Wu Zhou[5]
Sencun Zhu[2], Peng Liu[2], Suresh Chari[1], Ce Zhang[6]
IBM T.J. Watson Research Center[1]; The Pennsylvania State University[2]; Georgia Institute of Technology[3]
The University of Texas at Dallas[4]; North Carolina State University[5]; ETH Zurich[6]
hhuang, schari@us.ibm.com[1]; *sxz16, pliu*@psu.edu[2]; *cong*@gatech.edu[3]; *ce.zhang*@inf.ethz.ch[6]

***Abstract*—Android malware scanning services (e.g., VirusTotal) are websites that users submit suspicious Android programs and get an array of malware detection results. With the growing popularity of such websites, we suspect that, these services are not only used by innocent users, but also, malware writers for testing the evasion capability of their malware samples. May this hypothesis be true, it not only provides interesting insight on Android malware development (AMD), but also provides opportunities for important security applications such as zero-day sample detection. In this work, we first validate this hypothesis with massive data; then design a system *AMDHunter* to hunt for AMDs on VirusTotal that reveals new threats for Android that has never been revealed before. This is the first systematic study of the malware development phenomenon on VirusTotal, and the first system to automatically detect such malware development cases. AMDHunter has been used in a leading security company for months. Our study is driven by the large amount of data on VirusTotal—We analyzed 153 million submissions collected on VirusTotal during 102 days. Our system identifies 1,623 AMDs with 13,855 samples from 83 countries. We also performed case studies on 890 malware samples selected from the identified AMDs, which revealed lots of new threats, e.g., the development cases of fake system/banking phishing malware, new rooting exploits and etc.**

***Keywords*-Android; Malware Development; Malware Analysis; Threat Intelligence; Machine Learning; VirusTotal;**

I. Introduction

Developing and testing of Windows malware through a *private* multi-scan system that contains up-to-date services of multiple antivirus engines is a new phenomenon in the malware research community [1], [2]. According to recent news [3], two English suspects were arrested for building a *private* multi-scan system for malware writers.

With the phenomenal growth of mobile users, its leadership position in the global mobile device market, and heavy customization by vendors that produces fragmented and vulnerable OS images, the Android system is an attractive target for malware writers. Consequently, significantly more Android malware has appeared and been easily spread [4], [5]. Indeed, one research study [6] has shown that, in 2015, over 95% of mobile malware targeted Android. In this work, we suspect that Android Malware Development and testing could also exist on various *public* scanning platforms (e.g., Malwr [7], VirusTotal [8]). We conduct the first systematic study and develop the first scalable system to automatically discover such development patterns, namely, the **Android Malware Development** (the AMD) patterns.

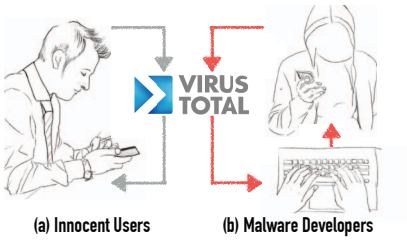

Figure 1: Malware development patterns are buried in millions of VirusTotal submissions.

To study the interesting AMD phenomenon and understand emerging trends of Android malware threats, we carry out a large-scale systematic study to reveal the AMDs on the VirusTotal platform,[1] which we have called *AMD hunting*. VirusTotal is the largest, and leading, malware submission and scanning platform used by many top security companies and the malware research community.[2]

Study Highlight. During our study period (10/05/2015–01/15/2016), there were about 1.5 million suspicious file submissions in VirusTotal per day, among which about 500K were Android-related (e.g., *.apk*) files. For all the Android-related file submissions, more than 55 antivirus engines provide malware scanning and detection. The identified malware samples are labeled by each tool and returned as a report. From mining this massive data set, we try to reveal an interesting phenomenon: this free service is used as a convenient channel for malware developers to easily test their samples before spreading them to distribution channels. In this case, malware developers submit their developed malware, and keep twisting their malware based on the returned reports. We built a system to automatically detect this phenomenon. This allowed us to identify 1,623 AMD

[1]Our research focuses on VirusTotal as a case study, but the designed approach can be adapted for hunting the aforementioned malware development cases on similar platforms (e.g., Malwr [7], ThreatExpert [9]).

[2]With the help of VirusTotal [10], malware researchers identified the malware samples that were used to hack Sony Pictures.

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

cases across 83 countries, 64% of which were not detected by any antivirus engine; we estimate that at least $> 90\%$ of them were indeed malicious.

Challenges. To automatically identify AMD patterns on VirusTotal, we faced two major challenges.

Challenge I, Scale: VirusTotal has more than 1.5 million submissions per day, and thus downloading all the Android relevant samples and performing detailed analysis is not practical. Hence, we needed to design a scalable, efficient, high-recall, and near real-time system to filter suspicious AMDs *based on metadata* without heavy malware analysis.

Challenge II, Validation: Given suspicious AMD samples, validating them entails a lot of background noise (e.g., unknown file types, random customer submissions) and limited supporting evidence. Therefore, it is very difficult to *validate* these suspicious AMD cases.

Motivated by these two challenges, we decided to perform a two-phase analysis. *Phase 1* carries out a pre-filtering process by only using the metadata to capture suspicious AMDs in real time (e.g., submission *ID*, *time*, *country*, etc.). We show that a very simple classifier is able to conduct this phase with properly engineered features. *Phase 2* performs an in-depth analysis on the corresponding samples from the suspicious AMD submitters filtered by Phase 1. In this phase, we designed validation logic to validate the suspicious submitters, which is based on our understanding of Android malware, the nature of AMDs, and fine-grained malware analysis techniques. We performed package level, static code level, and dynamic runtime analysis to collect evidence among samples from every suspicious submitter. For the malware analysis result, we then performed several similarity measurements to help distill and render all the results in a meaningful way (e.g., runtime result comparison of samples, similarity based grouping of submitters, etc.).

Technical Contributions. Our contributions are threefold:

C1: New Hunting and Validation Approaches. We designed a scalable hunting framework to distill out suspicious submissions from VirusTotal and perform validation on the suspicious submitters. We designed an approach to validate the suspicious AMDs based on Android malware analysis techniques (e.g., package-level analysis, static analysis, dynamic analysis) and similarity measurement.

C2: New AMDs, Malware Analysis and Case Study: We identified and analyzed 1,623 AMD cases across 83 countries with a total of 13,855 samples. When first submitted to VirusTotal, 64% of them were not detected by any antivirus engine. We then selected 10% of these undetected samples for detailed analysis and confirmed that *all of them* are indeed malicious. We performed further case studies in order to report interesting AMDs to the security community via direct intelligence sharing or industry reports.

C3: Implementation. We implemented our framework in Spark, which is able to continuously process 153 million VirusTotal submissions. This system has already run continuously in use by a leading security company for four months.

Related Work. Large scale malware study has been performed on VT dataset [11], [12]. Kantchelian et al. [12] leverages various supervised and unsupervised techniques to help aggregate various labels from different AV vendors into one ground true label on the VirusTotal dataset. However, **Malware development and testing study** [1], [2] has only been carried out for Windows binaries. Android malware is quite different—for example, the repackaging technique is rare in Window binary but very popular in Android APKs. Also, unlike the earlier study [1], we conducted more systematic research for malware development hunting and added an automated validation phase. The approach described in [2] for mining Windows malware development in Anubis requires all the detailed static and dynamic analysis results to perform classification. It also requires an external dataset from Symantec WINE for validation. In contrast, for our research, we as a third party do not have access to detailed analysis results from VirusTotal, and our validation logic is based on the understanding of Android malware and AMD cases, which is very different from Windows binaries.

Android malware detection [6], [5], [4], [13] have been proposed for app markets. Zhou et al. [13] conducted the first systematic study of Android malware. However, we focus on the discovery of such malware, and therefore the 13,855 samples we discovered, can serve as a fresh dataset for advanced Android malware research, which has also been demonstrated in our case study. Traditional malware analysis/detection [6], [4] focuses more on choosing the right detection heuristics, while **AMD hunting** can support malware detection, they are dramatically different. Malware detection usually takes place immediately after a piece of malware is distributed onto a user's device or after it has been widely distributed; in contrast, in AMD hunting, by identifying the suspicious submitters, one can directly learn the malware writers' developing techniques and be alerted of new malware variants even before they are spread. Moreover, one can find 0-day Android malware components that are under development, which can be later used as payloads for malware droppers. Empirical security studies [14], [15], [16] on the mobile system/applications and its ecosystem [17], [6], [18] have been performed previously. While our AMDHunter at this stage focuses more on revealing the AMD phenomenon efficiently and effectively, which initiates several new research directions, e.g., the scalable study and analysis of the malware development phenomenon and big data security research. To better analyze the malware, we adopted various similarity measurement techniques [5], [6] for AMD validation and SubID grouping, which reveals malware components that are still under development.

II. Background and Definitions

Our study is the largest to date on a third-party-collected online malware submission dataset. In this section, we describe the characteristics of our data set, and introduce definitions and other terminologies.

A. Definitions and Terminologies

Figure 2 illustrates the VirusTotal (VT) ecosystem and an example of the metadata that we collected:

Submitter is a user who submits files/URLs to VT and has a unique hash value (e.g., *aefd04ad*) as the submitter ID (SubID). SubID is computed based on some contextual details of the submissions and a dynamically generated salt to ensure the contextual data cannot be reversed. If no account is generated, VT uses the IP address to derive the SubID. Otherwise, the SubID is generated based on more detailed user account profile. Submitters can use the web portal of VT or provided APIs to submit.

Sample is a suspicious malware-related file submitted by a submitter. In the context of this study, a sample is an Android related file. When a sample is confirmed to be malicious, we call it malware, which can be small components of malicious applications (e.g., droppers, unpackers and rooters etc.). In this paper, we use first five characters in a sample's sha256 are used to refer to it (e.g., 1234e***), and sometimes also with SubID (e.g., (aefd04ad, 1234e)).

(Submission) Trace is a set of submissions from the same user (SubID), ordered based on submission time. Within a *trace*, multiple submissions of the same sample is possible and each *submission* is uniquely identified by *(SubID, sha256 and submission-timestamp)*. If the trace contains an evolving malware family and shows at least one occurrence of reduced positive number on some pairs of malicious samples, we call it an *evasive trace*.

Positive Number is number of antivirus engines that label a submitted sample as malware. The *reduced positive number* is a relation between a pair of functionally similar samples: when they are submitted by the same user (SubID), the later sample has a lower positive number than the earlier one. Note that one sample can be analyzed at different times, so the returned positive numbers can vary. Here, we calculate the *reduced positive number* based on the returned positive number when samples were submitted by a specific user.

The goal of our study is to identify *Android Malware Development (AMD) Cases*—Android malware writers might leverage the convenience of VT as the platform for malicious components or malware testing, which we call *an AMD case*. Each AMD case maps to a unique user (SubID).

B. Data Collection and Statistics

We collect about 1.5 million submissions from VT each day, through the VT privileged API provided by a security company. We have continuously processed the submission metadata for 102 days (from 10/05/2015 to 10/15/2016). We have identified 978,447 unique SubIDs during this period and 85,899 from them have Android relevant submissions. Every minute, our system receives about 1050 submission metadata worldwide. On average, each SubID submits about 156 times. The submission metadata (on the right side in Figure 2) only contains the relevant information about each sample submission, which however does not include malware analysis details (e.g., static/dynamic analysis result).

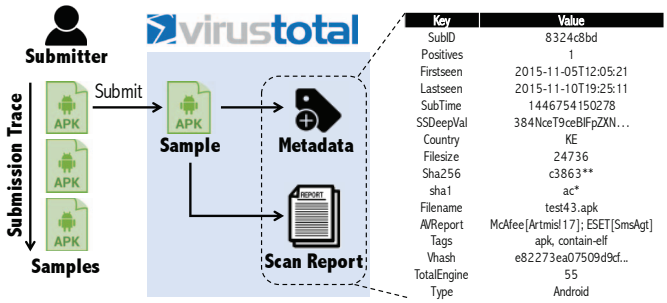

Figure 2: Illustration and Terminologies of VirusTotal.

III. AMD Hunter

We explain the design and the implementation of our *AMDHunter* within the framework shown in Figure 3. AMDHunter has two primary phases: *the AMD alerting phase* and *the AMD validation phase*. This two-phase design is tailored specifically for handling the huge data volume challenge—the processing speed should match the average data stream of around 20 submissions/sec from VT.

In *Phase I*, AMDHunter takes as input a stream of submissions, and outputs submissions that are *suspicious*. This is a classification task, and we develop a lightweight classifier to spot the potential AMDs. In this phase, we only use the submission metadata (in Figure 2) to avoid expensive and detailed malware analysis for all submissions.

In *Phase II*, all suspicious samples from the last phase are downloaded and analyzed. The goal is to conduct detailed analysis to spot real AMDs from these samples. We implement and deploy three types of analyzers: *Package Analyzer*, *Static Analyzer* and *Dynamic Analyzer*, to extract and analyze various kinds of characteristics of apps. The analysis results are collected, parsed, and encoded into multiple supporting evidences. Then, our *(Suspicious Submitter) Validator* determines AMDs, based on multiple evidences.

After *Phase II*, we group the AMDs based on the sample similarity and perform 0-day malware confirmation for the 0-positive-number samples. We then render some of the reported AMDs to malware analysts from a leading security company for final validation. In the rest of this section, we will introduce the entire workflow step by step.

A. Phase I: AMD-Alerter Design Logic

Our AMD-alerter is basically a scalable classifier, which classifies traces into *suspicious* or *normal* in real time. Given the fuzzy nature of reasoning the AMDs, in our system we

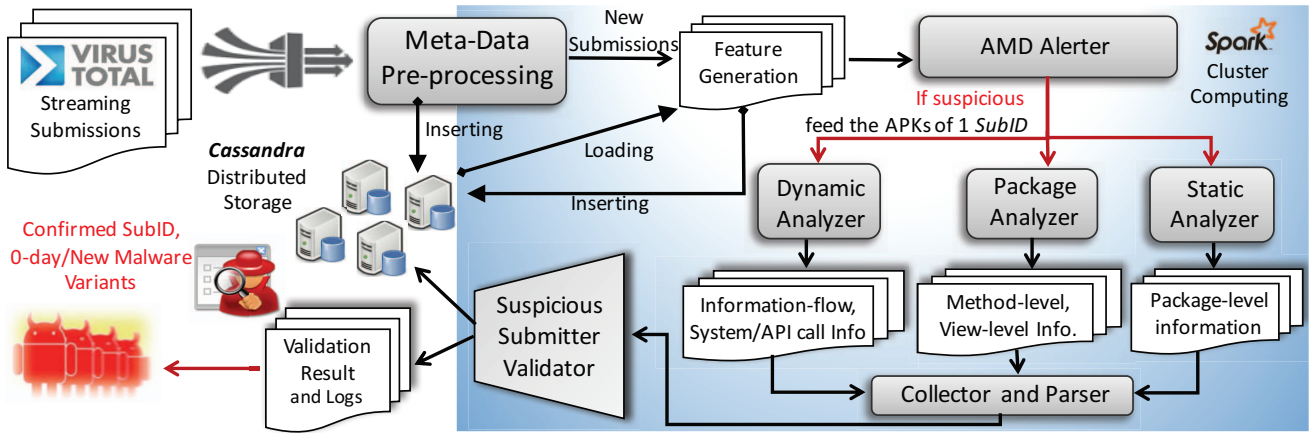

Figure 3: The AMDHunter Framework

measure the likelihood of a trace being an AMD case with the *Bayesian* classifier. Specifically, whenever a new submission comes, we check if its SubID exists in our database; if existing, we load all the previously recorded submissions from that SubID and then compute a *feature vector* (FV) for the SubID. The FV is further fed into the Naive Bayesian (NB) classifier for suspiciousness measurement.

For scalability and real-time processing, initially our AMD alerter only leverages the submission metadata information. The question is: *what are the predictive features for AMDs in submission metadata?* Our classifier may directly use all the fields of the submission metadata from each SubID as its features. However, without using more semantics-relevant features, the classification can hardly be *robust* and *predictive*. Therefore, we perform feature engineering and extract a set of derivative features from the fields in the submission metadata. All the features are computed based on submission *traces* from each SubID. Our classifier will then determine the suspiciousness of a SubID based on the *whole FV* but not on any single feature f_i. Due to page limits, we only explain some of derivative features for classifying suspicious/normal SubIDs:

f_1: When most of the submissions in a trace have a submission time later than the first-seen timestamp on VT, that is, when most of the samples from this SubID have been previously submitted by other users on VT, then it is likely not a suspicious AMD case. This is because malware writers are not likely to resubmit identical samples. Most of the samples from AMDs should be newly manipulated or developed and hence have unique hash values.

f_2: Within a trace, if the file names of some samples contain compiler generated or suspicious key words (e.g., $test_x$, *debug, signed, aligned* and etc.), the samples are considered as suspicious. Because some Android app building tools use automated naming schemes during compilation.

f_3: Within a trace, if the SubID contains suspicious tags, then we consider it as suspicious, e.g., the "via-tor" tag indicates that it is submitted via the *Tor network* anonymously.

f_4: Within a trace, if there are some reduced positive numbers among samples with similar file names (e.g., those with shared subsequences), it is a suspicious trace.

f_5: Within a trace, if the SubID contains lots of different types of executable files or different file types, then we consider it a normal trace. This is because malware writers usually focus on a small set of file types (e.g., *apk, dex* or *elf* for Android).

We model some of these features as continuous variable and use the Gaussian distribution [19] and some of these features as Boolean variables. For training, we use `3k` submission traces as our training set from previously collected SubIDs from study [1] and some manually labeled SubIDs by malware researchers from a leading security company.

B. Phase II: AMD-Validation Design Logic

After the AMD alerter spots a trace for a SubID as suspicious, we will analyze all the sample files from the suspicious SubID. As shown in Figure 3, our analyzers will extract package-level information, static code information and dynamic code behavioral information on each sample and then generate various "supporting evidences". All these "supporting evidences" and intermediate analysis results are then gathered, parsed and finally fed into the *Suspicious Submitter Validator* for decision making according to multiple pieces of evidence. In the following, we will elaborate on the supporting evidences (***E***) and some details of the analyzers.

1) Package analyzer and supporting evidences: Our package-level analysis relies on several unique observations of the Android app developing process. We find that in the current Android compilation process, the *aapt* (i.e., Android Asset Packaging Tool)[3] *only* updates the timestamps of the recently modified files in the application package; that is, it will not change the timestamps of all the other unmodified files. Normally, neither will a malware tester manually change the timestamps of the files after compilation. These observations help us perform several forensics.

[3]The APK file format is quite similar to zip format, one can simply use *unzip -l* to collect all the time stamp information and file names.

E1: When the timestamp of any file in the APK package is close (e.g., $<$ 24hrs) to the submission timestamp on VT, we consider this a supporting evidence. The app is unlikely submitted by a user/security researcher who caught it in the wild, because it takes time for a new app to be distributed, get downloaded and further trigger some security alerts.

E2: When samples from a SubID share lots of identical timestamps, it is a supporting evidence. This is because a malware writer may tune a sample to make different versions of the package APK files. During the process, most of the files (e.g., the resource files) in the APK keep untouched.

E3: When the timestamps of all the files in the APK file are identical, we will report it as a supporting evidence. After all, having identical timestamps for all files in an APK is abnormal, as the code files (e.g., .dex, .os) should have different timestamps than other resource files.

E4: Each Android app needs to be signed with a certificate, and the information of every signing META-INF/CERT.RSA of the app when compiled. Android uses it for app integrity and permission checking. By using the *keytool -printcert* command, one can extract the detailed certificate information. If a set of apps in a suspicious trace is signed with the same certificate, they are probably from the same author. Hence, it is very likely an AMD case. To reduce false positives, we whitelist some known certificates (e.g., the development keys provided by Google, which has been used by many newbie developers).

2) Static analysis and supporting evidences: **E5:** A malware writer may keep on modifying and testing malware samples without changing the primary UI of the app. If a set of apps from one trace share similar UI structures, then it is very likely that the alerted trace is an AMD case. To measure UI structure similarity, we use the view graph approach [5] that is designed for app repackaging detection.

E6: method-level similarity. Sometimes a malware author will test a set of samples that are repackaged with different carrier apps but share similar malicious components, e.g., insert an identical Trojan component into various fake system apps. Through the method-level comparison, we can identify those malicious components with similar control flow graph structures (CFG). For this purpose, we deploy Androguard to perform static CFG construction and use the centroid construction approach described in [6]. If the similar methods are used by a set of apps in a trace, then this trace has a supporting evidence (E6).

E7: Runtime similarity. Some samples in a trace might have been heavily obfuscated or packed with various packers (e.g., 360 packer and bangcle) or other anti-static analysis techniques; hence, we also measure their runtime behavioral similarity. Whenever we identify a pair of Android samples in a trace yielding similar runtime behavior, we will consider the trace a candidate AMD case. We measure this aspect of similarity using a set of specific runtime features we collected from a specific analysis engine, which we built based on an existing runtime analysis platform, namely the *Droidbox* [20]. Our customized *Droidbox* can collect 35 general runtime features. For example, the features include various suspicious Android APIs (e.g., *android.webkit.WebView;→loadData*), system calls (e.g., futex calls), sensitive taint flows, and code loading behaviors (e.g., *dalvik.system.DexClassLoader;→loadClass* used in *com.qihoo.sec.lib.resl.ExtRes* by the 360packer)).

3) Validation Logic: After all supporting evidences are collected for a trace, we can then make a validation decision based on them (e.g., E1-E7). We summarize our validation criteria as first-order logic. The validation result (VR) is formalized as the following formula with basic predicates e_1 to e_7, each representing the fact that the corresponding supporting evidence is *true*:

$$
\begin{aligned}
VR(v) &::= v_1 \vee v_2 \vee v_3 \vee v_4 \\
v_1 &::= e_1 \vee e_2 \\
v_2 &::= (e_4 \wedge e_3) \vee (e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7) \\
v_3 &::= (e_5 \wedge e_6) \\
v_4 &::= \neg(e_5 \vee e_6) \wedge e_7
\end{aligned}
$$

Overall, we have four validation ***rule***s (from v_1 to v_4) based on various heuristics. Note that here **disjunction** (instead of conjunction) is used between ***rule***s, so as long as any one of the rules is validated, the SubID will be confirmed as a malware writer, which helps cover various cases. The first rule (v_1) states that if either the submission time is very close to the compilation time (E1) or most of the files have shared timestamps, we consider the trace an AMD case. This rule captures the cases when the malware testing samples are submitted within a short time period after compilation or when the malware writer keeps on testing a series of versions for one malware. The rule v_4 help find stealthy AMDs that contain packed/or highly obfuscated versions of previous malware. Basically, within a trace if some samples have no static similarity (E5 or E6) but have dynamic behavioral similarity (E7), it is very likely some type of anti-static analysis techniques are deployed. Using this rule, we have identified lots of traces with commercial or home-brewed anti-static analysis techniques. For instance, a submitter (*c555bc30*) used a commercial packer (i.e., *360-jiagu*) from China, and another submitter (*c9e0b761*) designed a home-brewed steganography technique in sample *4a143*. Note that the supporting evidences and validation rules proposed here are not meant to be complete. New rules can be added into the system from study of confirmed AMDs.

IV. System Implementation

Now, we present the system implementation of our AMD-Hunter, shown in Figure 3. The current system is built upon a high performance cluster of 20 nodes, with the following system and hardware specifications: Linux 3.16.0-4-amd64,

32G RAM, 8 core CPU (freq: 800MHz) for each node and we use YARN and Mesos for cluster management. The AMDHunter is primarily implemented in Scala, the native language that is used to design the Apache Spark. We wrote 3,894 lines of Scala code to build distributed processing jobs, 2,330 lines of python scripts and 692 lines of shell scripts to perform further data analysis when major tasks are finished and result is collected to the master node. The general workflow has 3 steps, *First:* meta-data pre-processing and feature generation, *Second:* suspicious AMDs alerting via the *alerter*, and *Third:* AMDs validation by the *validator*.

Preprocessing and feature generation. To handle the large volume of streaming submissions (average 40 submissions/sec, the peak can be 150 submissions/sec), designing and implementing an efficient meta-data pre-processing and analyzing system is very critical. In our current system design, we leverage the powerful Spark [21] distributed computing framework to parse the meta-data and feature generation for each submitter (either generating the features of a new trace or updating the old traces in the database). In addition, we choose the Apache *Cassandra* [22], which is an open source distributed database management system. The *Cassandra* is designed to handle large amount of data across commodity servers. Besides high availability, low latency for data retrieval and proven fault-tolerance, it provides a language named *Cassandra query language, a sql-like language*, which helps us analyze the data based on various flexible querying statements. This is critical for our initial stage data exploration tasks. To optimize the throughput, we provide several "lazy evaluation" criteria for the feature generation tasks. For instance, we will not generate the feature of a SubID until we see a submission with a file type in the trace matching the executable binary types (e.g., *apk, dex, elf, jar, peexe, maco, and etc.*) of our interest.

AMD Alerting: Naive Bayes (NB) Classifier in Spark-ML. Since the NB is a simple multiclass classification algorithm[4], which is based on the Formula 1:

$$Pr\left(Y = y_k | F_1 ... F_n\right) = \frac{Pr\left(Y = y_k\right) \cdot \prod_i Pr\left(F_i | Y = y_k\right)}{\sum_j Pr(Y = y_j) \cdot \prod_i Pr\left(F_i | Y = y_j\right)} \quad (1)$$

The NB classifier can be trained very efficiently using one spark task. Whenever a new SubID is processed with a new/updated feature set, we run our newly trained model on that feature set, and then the result (i.e., suspicious or normal) returns immediately. The model is trained and runs at the spark Machine Learning framework (*spark.MLlib* [23]). According to the nature of our problem and the designed features, we set the model type as "Bernoulli". For every 100 new identified AMDs, we retrain our model. The AMDs are labeled by the *validator* and confirmed by malware researchers. We also tested with other classifiers in the *spark.MLlib* (e.g., Support Vector Machines (SVM)) with similar accuracy to the NB Classifier but higher overhead in the training phase. Since scalability is our first priority in this AMD hunting project, the NB Classifier is a better fit at this stage. Other classifiers and corresponding optimization could be tested and easily plugged into our framework.

AMD Validation: Parallel sample analysis in Spark. After the alerter raises an alert on a suspicious trace, we schedule multiple Spark tasks for multiple analysis components (i.e., package-level analysis, method-level analysis and view-level similarity analysis). Since all the analysis components are developed in python, we use python scripts to glue them and call the Spark task scheduler. For the runtime-similarity analysis, we use running the emulators that we built based on *droidbox*, and the generated results are saved in database and retrieved by a dedicated Spark task. After the detailed analysis (static similarity measurement, runtime similarity analysis and etc.) from all the components is done for each suspicious trace, we use the *Spark collect()* method to aggregate detailed result across all the nodes in the cluster to one single node to compute the supporting evidences accordingly to the validation rules. For instance, if the runtime analysis component reports that samples have a high runtime similarity (E7 is true), however, either E6 or E5 is reported false, then the aggregated result will return a matching against *rule 4*. Also, the detailed analysis information in the Spark cluster is directly inserted into predefined *Cassandra* tables with indexes for further analysis.

V. AMD Hunting Result

The hunting system was built and launched on Oct. 5th, 2015 in a leading security company. After 102 days, our system streamed and stored a total of 153 million of submissions from VirusTotal (VT), and the classifier evaluated all the SubIDs. Next, we will report the main results. Some additional results can be viewed in the **result website** [24].

A. False Negatives and False Positives

Till Jan 15th, 2016, our *AMD alerter* reported 3,078 suspicious AMDs out of 85,899 SubIDs.Then our *validator* automatically confirmed 1,625 of the 3,078 traces as AMDs. We grouped 234 (out of 1,625) SubIDs into 78 clusters based on similarity measurement of SubIDs (details in Section V-B), and all the rest 1,391 SubIDs did not form any cluster. For further manual examination and detailed case study, we picked 2 SubIDs from each SubID cluster and randomly picked 250 SubIDs from the rest SubIDs to form a set of 406 SubIDs (about 25% of the 1,625). We only found two cases were false positives. One is a greyware tester, which provides automatic gaming points collection functionality in a cracked and repackaged game (SubID : *17378139* from Taiwan). Some samples from the other case have been labeled as a potentially unwanted program by some AV engines, which is a chat app named **galaxy app**

[4]For our Bayesian classifier, we assume conditionally independence between every pair of features.

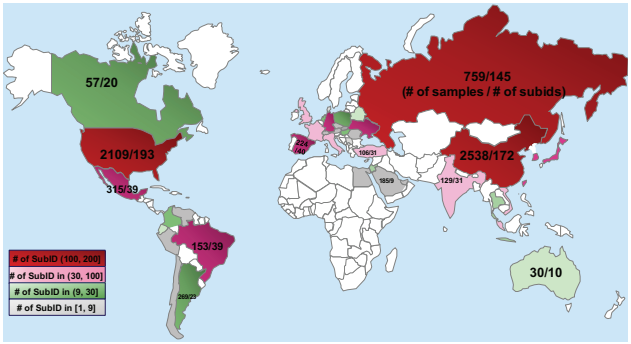

Figure 4: The 1,623 AMDs across 83 countries

from the SubID *f6b47657*. The developer tested different versions of the app on VT.

Based on the above sampled verification, the false positive rate of our AMDHunter is 0.49%. This demonstrates that with the *validator*, our system can report the AMDs at a very low false positive rate. We also clustered the 1,451 non-validated cases from 3,076 suspicious AMDs and then similarly picked 25% from the set. We found no AMD. This indicates that our system has a very low false negative rate. All the confirmed AMDs are then used for a final-step double examination and analysis by several malware analysts. We also reported many zero-day malware samples to the VT community after March 2016.

B. General AMD Analysis

We further performed analysis on the 1,623 identified AMDs, regarding submission location distribution, sample-similarity based SubIDs grouping, evasive traces, 0-day samples and AV reactions.

1) AMD Location Distribution: The 1,623 submitters were distributed across 83 countries with a total of 13,855 malware samples

From the distribution map in Figure 4, we observed that most of the submitters and the samples were from the US, China and Russia. China had the largest volume of submitted malware samples and US had the largest number of AMD submitters. The other countries with most submitters were Mexico, Brazil and several European countries. When comparing the result with a report [25] from Mcafee, we found that the countries with more malware writers did not necessarily have more malware detections/infections. For instance, our study revealed that the country with the largest number of malware submitters was not China but US, but in the report [25], it was found that China had the largest number of malware detected and India had the largest number of malware infections. The average numbers of samples per AMD submitter were quite close in different regions, mostly around 10 malware samples/submitter and no more than 25 samples/submitter. One interesting case was SG (the Singapore region), which had an average of 74.7 submissions/submitter. Later, our sample-similarity-based SubIDs grouping helped us identify a malware writer (with several SubIDs) from a famous university in SG, who used 13 accounts to perform several batch submissions of evolving samples. Later, we found the SG submitter is a penetration tester who is performing evasive AV engine testing on VT and reporting their findings [26] in a premium security conference, namely AsiaCCS 2016.

2) Sample-based Similarity Grouping of AMDs: Another interesting analysis we have done is to group the identified SubIDs (1,623) into clusters based on their sample similarity. Specifically, we made pair-wise comparisons of samples from different SubIDs based on (F1) runtime FV, (F2) static view structure, and (F3) static method. This process was done efficiently (in less than 1 hour), as we have all the intermediate results previously computed and we did a lazy evaluation for these three ordered similarity measurement. For each pair of samples from different SubIDs, if (F1) measured and returned as similar, we would not further evaluate (F2) and (F3). Based on such pair-wise similarity measurement among samples, we then performed the grouping for the SubIDs. Whenever there is a pair of SubIDs (A, B) that share more than 40% of similar samples (i.e., # of shared similar samples/*MIN*(A, B)), we added them into a similar group. We performed this for all the SubIDs until all of them were processed and put into groups. We find that a total of 234 (14.4%) SubIDs are forming 78 clusters, from which we randomly analyzed 20 groups and found all of them were developing Android malware with close functionalities, including malware SDK developing the aforementioned Ph.D students from the Singapore, Bank Trojans, etc. This implies that some of the malware writers are using multiple accounts or IPs for their submissions.

3) Evasive Traces: One main reason for the existence of AMDs is that malware writers keep on changing the malware samples so that the later version could evade the detection of the latest antivirus detectors. We defined a submission trace with such a pattern as an *evasive trace* in Section II. Hence, identifying and understanding these *evasive traces* can help understand the weaknesses of antivirus engines. For each SubID from the 1,623 confirmed traces, we further put all the samples in a trace into different sub-similarity clusters (based on static and dynamic similarity measurement described in Section III-B). We ordered the samples in each sub-similarity cluster based on their submission timestamps and their corresponding positive numbers, and then checked if the trace is an evasive one. Figure 5 shows all 99 identified evasive traces. The x-axis shows the corresponding SubID *and* the number of versions it submitted for the reduced positive number, and the y-axis is the positive number. The upper line connects all the original positive numbers and the lower line connects all the reduced positive numbers. For example, in the 9d5e7d11-[2] case, the SubID is *9d5e7d11*, "[2]" indicates that 2 extra versions are submitted for testing, and positive number is reduced from 39 to 21. We can observe from the figure that 45 traces have been reduced

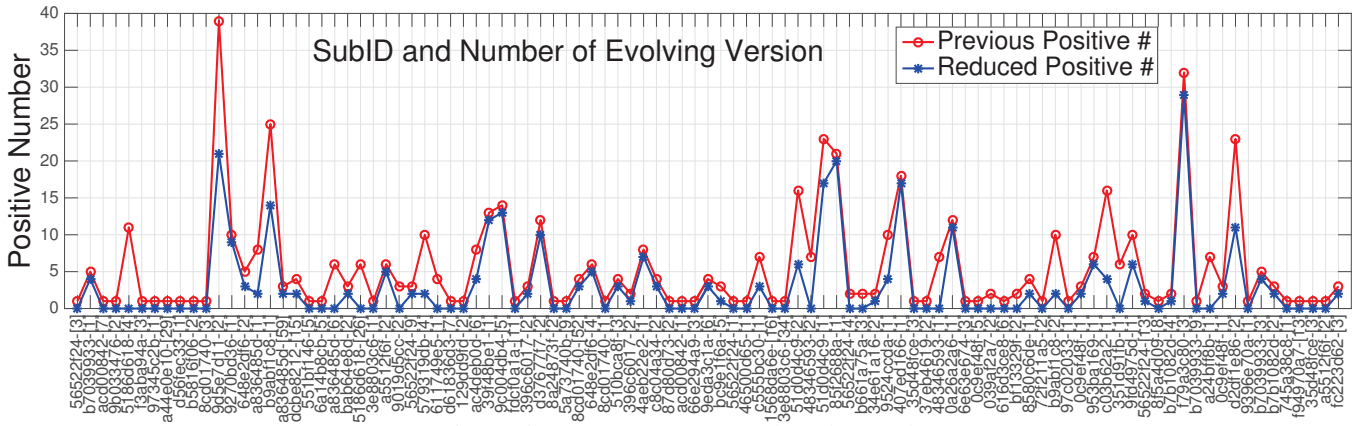

Figure 5: *Evasive AMD cases* based on SubIDs

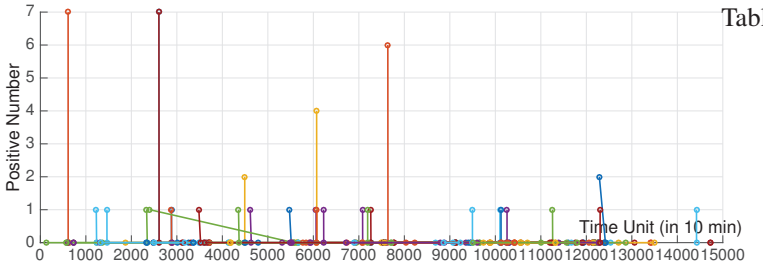

Figure 6: *Evasive submission traces* spans 102 days

Table I: The distribution of positive numbers (PosNum)

Result from 58 vendors	Percentage	# of Samples
PosNum: 0 vendor	63.75%	8,833
PosNum: 1 vendors	9.77%	1,354
PosNum: 2 vendors	3.77%	523
PosNum: 3 vendors	2.63%	365
PosNum: 4 vendors	3.59%	497
PosNum: 5-10 vendors	4.53%	628
PosNum: 11-20 vendors	5.38%	746
PosNum: 21-30 vendors	5.90%	818
PosNum: 31-60 vendors	0.66%	91
Total of 58 vendors	100%	13,855

to 0 and only 3 traces had original positive numbers larger than 25. This indicates that most of the evasive AMDs that we identified are not widely-recognized by the AV engines. We further checked all the traces that were reduced to 0 by drawing them on a 102-day timeline (i.e., 15k * (10) minutes on the x-axis of Figure 6). From the figure, we can see that 26 of the AMDs have a steep-down line, as they are submitted within a very small time window (around 20-30 minutes). This indicates that some mature tools or techniques have probably been adopted by malware writers to perform malware manipulation, obfuscation or packing, etc. Also, based on our observation, very few of the evasive AMDs have a period of more than a month. Our conjecture is that malware writers probably keep switching account, e.g., one VT account for a project, which matches the result in the SubID grouping analysis.

We picked some of them for further analysis and made several new findings about Android malware, including steganography techniques used for payload hiding, building malicious code logic into JavaScript for evasion and etc (details in Section VI).

4) Zero-day Variants Analysis: Table I shows the distribution of positive numbers based on first-time submissions, as recorded in our system (some samples were submitted and re-scanned several times with different positive numbers). For all the 13,855 samples from the 1,623 submitters, a large portion (63.75%) of them are 0-day or new malware variants with 0 positive numbers.

We randomly selected about 10% (890) of these 0-day (variants) samples for further validation by malware ana-

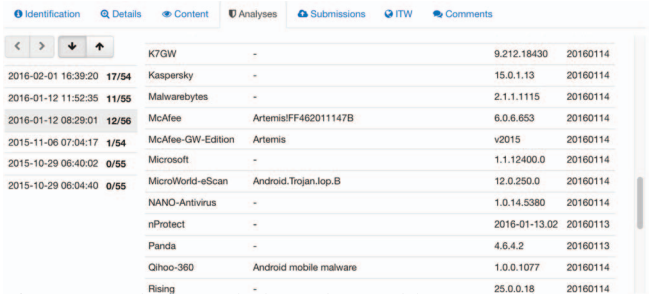

Figure 7: 0-day sample later detected by AV engines on VT

lysts. The validation process has two steps. First, whenever a pair of samples from the same SubID have (method-level/runtime) similarity, we will temporarily use the label from the known malware for the unknown variant. This helped us confirm 432 malware variants. For the rest 458 samples, a further similarity measurement was done against a large dataset of known malware from the company. We were able to confirm that 320 from this dataset are also malware variants. For the rest 138, we performed manual checking and more detailed analysis to find the potential 0-day samples (e.g., new bank phishing samples, new fake system apps and etc.). Some selected analysis details are presented in Section VI. With the help of the malware analysts from the company, we were able to confirm all of them are true positive samples. The details of this dataset can be viewed in our *result website* [24]. At the meanwhile, we have notified most of the other vendors for the 8,833 undetected cases and they will further analyze these samples and share their insights with the community.

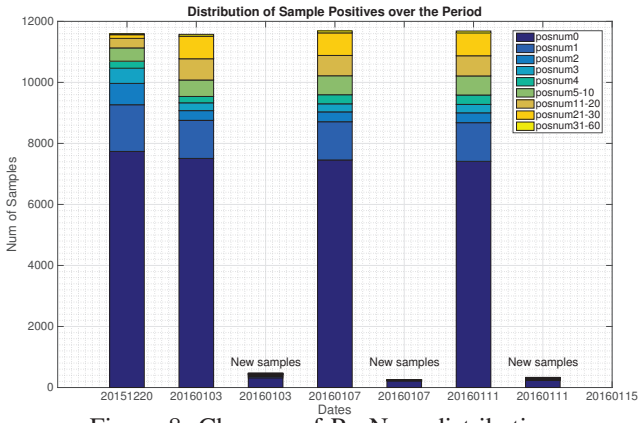

Figure 8: Changes of PosNum distributions

5) Changing Trend of Positive Numbers (PosNum).: To check how well the AV engines on VT have reacted to the set of samples we identified based on AMDs, we conducted a positive number evolving trend analysis starting from Dec 20th, 2015. In Figure 8, the 1st stack-bar (on the leftmost) shows the positive number distribution for all the samples (11,887) from the identified 1,239 SubIDs on Dec 20th, 2015. We then performed a re-scan on VT after two weeks (on Jan 3, 2016) for the same SubIDs and the same set of samples (i.e., 2nd stack-bar). Around 500 new samples from these SubIDs were identified (depicted in the stack-bar right next to it), and it shows that most of them had 0 positive numbers. Then we performed the same analysis for the 1,239 SubIDs on the 7th and 11th of Jan, respectively. From the four high stack-bars, we see the trend that most of the non-positive samples are updated with a larger positive numbers, which means more vendors will be able to label the "detected" samples by following other vendors' label or update their signature database accordingly.

While some of the 0-day samples were detected by a few AV engines, they only accounted for a small portion (less than 500 out of the whole 7,789) during the three weeks period. In other words, the current AV engines cannot yet detect most of them. In fact, among the aforementioned 890 0-day samples we confirmed previously, 198 were used in this experiment and only two of them were later detected by some AV engines. The changes of engines' scanning results for the two are shown in Figure 7. Most of the 0-day samples that we re-scanned are still shown zero positive numbers in the end of Jan, 2016 This clearly signifies the early warning capability of our system. After we reported the results to other AV vendors for their signature patching in Feb, 2016, we stopped further re-scan. What's more, more 0-day variants have been identified based on our further tracking of the AMD cases.

VI. Case Study

In this section, we will elaborate on some of the findings based on our case study for the AMD hunting.

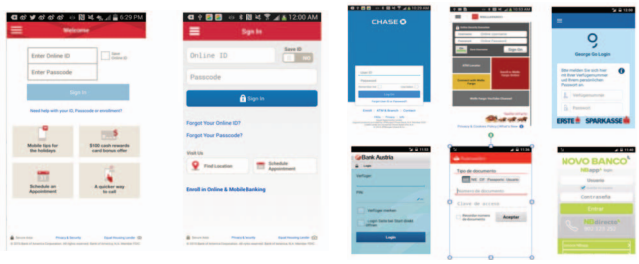

Figure 9: The fake log-in UIs for BoA and other banks

A. New Phishing Development Techniques

Phishing has been a severe threat against online activity, our study reveals a new form of phishing [27] targeting mobile users in the real world malware.

We initially identified and validated a trace (*4746befb*, Ukraine) through the supporting evidences *E1* and *E2*, which mean that the malware sample compilation timestamps match well with the submission timestamps, and most of the timestamps among the samples are identical (basically a continuous developing case). The phishing apps in the trace monitors the status of the foreground activity, phishes for and harvests authentication credentials when some specified target (banking) app is launched in the foreground. We find that samples from this trace target Australia, Singapore, HongKong, US and etc. (shown in Figure 9).

With the *AMDHunter*'s continuous tracking capability, we later identified more samples from this submitter and other submitters of this kind (e.g., *c9e0b761, 8954e2e2*) via sample-based grouping of SubIDs. Our system finds that they all shared the same code logic. After further validation, we confirmed that they were using the same log-in UI hijacking code, but targeted at banking and financial institutions in other countries. The newly added mimicking banking log-in UIs for Bank of America (BoA) can be viewed in Figure 9 (left) and some more in Figure 9 (right). After our analysis, we found these samples evolved by adding more targeted mobile banking apps and more fake banking log-in UIs. Besides targeting mobile banking users, this malware family also steals the log-in credentials of other online services (e.g., Facebook, Whatsapp, Skype, PayPal and etc.). Thus, we have discovered a big ongoing malware campaign [28], [29] with new phishing techniques.

B. New Threats from Fake System Apps

A new trend of spreading Android malware persistently is to install apps into the "/system/app" directory by rooting the devices or embedding them into customized ROMs. To make it difficult to be found by users, such malware samples always use similar package names as system apps, for instance, "com.android.xxx". In our study, we discover several developing traces that perform rooting and stealthily install fake system apps, and these malware can also be embedded into customized ROMs for spreading. This causes a more persistent threat as users often consider the apps under system folders are from carriers and are risk-free.

For instance, our system reported a trace (*f9106582*). Two apps in the trace were submitted onto VirusTotal on 11/28/2015, with similar functionality but obfuscated to different fake system app names. They all try to get the root privilege and then to install the payload app into the "/system/app" folder. The relevant reverse-engineered code is shown in *figure 9* and *figure 10* (post-rooting operations) in our *result website* [24]. Our system identified this case based on the supporting evidence *E1* that their VirusTotal submission times are very close to the compilation times.

The other eye-catching AMD (*20221459*) identified by our system uses the *BaiduProtect* [30] techniques to pack one of the fake system apps, which roots the device. Our system identifies this trace based on our validation rule v_4. While all the static similarity support evidences return *false*, the runtime similarity measurement returns *true*, which helps confirm the trace as an obfuscation scenario.

Other new threats that AMDHunter discovered, including the development of new 1) hiding techniques for AV evasion, 2) root exploits, 3) JavaScript loading, 4) aggressive (ad) SDK (the details in [24]).

VII. Conclusion and Future Work

In this work, we have built a scalable malware hunting framework for VirusTotal, and have deployed it in a security company for 4 months. During this period, the AMDHunter has processed 153 million of VirusTotal submissions and helped identify and validate 1,623 AMD cases (with a FP rate of 0.49%), including 13,855 samples across 83 countries. We also conducted case studies based on detailed malware analysis of 890 0-positive-number samples. This study revealed lots of new Android malware threats, e.g., new phishing, rooting, AV evasions, JavaScript loading techniques and etc. Our research not only raises the awareness of the AMD problem itself, but provides a scalable framework to systematically study the malware development on virus submission platforms. Finally, the AMD cases and malware sample list will be shared with the research community.

We admit that more stealthy malware writers can change their submission patterns to evade our hunting or even build their own private multi-scan system [3]. We are aware of the fact that our AMD hunting is not complete and there are potential missed AMDs (e.g., we miss the cases when each distinct SubID only submits one version for VirusTotal testing). This limitation, however, can probably be overcome through traditional malware clustering techniques, based on runtime behavior or code similarity. Hence, we need to collaborate with VirusTotal to get the malware analysis result for every submitted sample in the future.

VIII. Acknowledgments

This work is supported by ARO W911NF-13-1-0421 (MURI), NSF CCF-1320605, NSF SBE-1422215, ARO W911NF-15-1-0576, NIETP CAE Cybersecurity Grant, NSF CNS-1505664, and NSF CNS-1422594. Any opinions, findings or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Science Foundation and Army Research Office.

References

[1] "VT Helps Hackers Attack You," http://goo.gl/BFwGE8.
[2] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti, "Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence," in *Usenix Sec '15*.
[3] "The two suspect arrested for running private multiscan system for malware writers," https://goo.gl/uiUa4R.
[4] 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 *NDSS'12*.
[5] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "View-Droid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of ACM WiSec '14*.
[6] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *USENIX Security '15*.
[7] "Malwr, 2010," https://malwr.com.
[8] "Virustotal," https://www.virustotal.com.
[9] "Threatexpert, 2010," http://www.threatexpert.com/.
[10] "Hunting Sony Hacking Malware on VT," http://goo.gl/e1h6HE.
[11] L. Song, H. Huang, W. Zhou, W. Wu, and Y. Zhang, "Learning from big malwares," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2016.
[12] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, "Better malware ground truth: Techniques for weighting anti-virus vendor labels," in *the 8th ACM AISec '15*.
[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *SP '12*. IEEE.
[14] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *the 22nd ACM CCS 2015*.
[15] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu, "Towards discovering and understanding unexpected hazards in tailoring antivirus software for android," in *the 10th ACM AsiaCCS, 2015*.
[16] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime," *Innovation in Mobile Privacy & Security IMPS'16*.
[17] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Trust and Trustworthy Computing*. Springer, 2013.
[18] Q. Guan, H. Huang, W. Luo, and S. Zhu, "Semantics-based repackaging detection for mobile apps," in *ESSoS '16*. Springer.
[19] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, 1995.
[20] "DroidBox," https://github.com/pjlantz/droidbox.
[21] "Spark: cluster computing," http://spark.apache.org.
[22] "The apache cassandra database," http://cassandra.apache.org.
[23] "Naive bayes - spark.mllib," http://goo.gl/nFQURB.
[24] "Result of AMDs," https://sites.google.com/site/amdhuntingresult/.
[25] "Mobile Threat Report, McAfee, 2016," http://goo.gl/VGr88Y.
[26] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *12th ACM AsiaCCS 2016*.
[27] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *SP, 2015*.
[28] "Bank Phishing Campaign: SlemBunk I," https://www.fireeye.com/blog/threat-research/2015/12/slembunk_an_evolvin.html.
[29] "Bank Phishing Campaign: SlemBunk II," https://www.fireeye.com/blog/threat-research/2016/01/slembunk-part-two.html.
[30] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *Computer Security–ESORICS 2015*.