AndrEnsemble: Leveraging API Ensembles to Characterize Android Malware Families

Omid Mirzaei♦, Guillermo Suarez-Tangil♣, Jose M. de Fuentes♦, Juan Tapiador♦, and Gianluca Stringhini♠
♦Universidad Carlos III de Madrid, ♣King's College London, ♠Boston University
{omid.mirzaei,jfuentes,jestevez}@uc3m.es, guillermo.suarez-tangil@kcl.ac.uk, gian@bu.edu

ABSTRACT

Assigning family labels to malicious apps is a common practice for grouping together malware with identical behavior. However, recent studies show that apps labeled as belonging to the same family do not necessarily behave similarly: one app may lack or have extra capabilities compared to others in the same family, and, conversely, two apps labeled as belonging to different families may exhibit close behavior. To reveal these inconsistencies, this paper presents AndrEnsemble, a characterization system for Android malware families based on *ensembles* of sensitive API calls extracted from *aggregated call graphs* of different families. Our method has several advantages over similar characterization approaches, including a greater reduction ratio with respect to original call graphs, robustness against transformation attacks, and flexibility to be applied at different granularity levels. We experimentally validate our approach and discuss three specific use cases: mobile ransomware, SMS Trojans and banking Trojans. This left us with some interesting findings. First of all, malicious operations in these types of malware are not necessarily exercised by using several sensitive API calls all together. Second, SMS Trojans have larger ensembles of API calls compared to the other types. Last but not least, we identified several samples with identical ensembles though being labeled as part of different families.

KEYWORDS

Android Malware, Malware Analysis, Malware Classification

1 INTRODUCTION

Android is the most popular Operating System (OS) worldwide, with a larger market share than Windows PC and Windows phone together [1]. Android is a complex system that can integrate third-party software from on-line markets, some of which are weakly vetted [2]. This altogether poses several security and privacy concerns. In terms of security, new attack vectors are discovered at an unprecedented rate [3], while, in terms of privacy, apps have access to a wide range of information they often collect in bulk [4]. Some Android apps also impact the integrity of web resources in a negative way by manipulating them in various ways [5].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329854

Malicious or potentially unwanted applications are programs purposely designed to attack the security and privacy of the devices and their users. Moreover, Android apps are commonly hardened with advanced anti-analysis techniques, including obfuscation [3, 6] and packing [7], which turn their analysis into a really challenging task. To cope with this challenge, Anti-Virus (AV) vendors and cyber security firms characterize newly discovered threats, label them with a *family* name, and share the specimens together with associated Indicators of Compromise (IoC) with the security community. Labels are usually assigned based on some static information, including code structures [8] and other IoCs which are easy to modify using different transformation attacks [9, 10].

Despite the importance of the labeling process, AV vendors use different criteria to name samples and families [11]. As a result, recent studies have shown that not all samples associated to a family are always related [12]. Furthermore, it is common to find the opposite: two apps in different families with related behaviors [13]. In addition, the majority of the labels assigned are not consistent with the actual behavior of apps [14] and, in most cases, each AV engine produces a different security report and risk score for a malicious application. Two main strategies are proposed to deal with these inconsistencies: i) considering sub-families (or *variants*) to divide families into smaller groups of apps with more akin behavior [13], and ii) extracting a unique *behavioral core* from each malware family. Both of these strategies suffer from important limitations as we explain next.

On the one hand, methods proposed for dividing families into sub-families are error-prone and inaccurate due to their dependence on either uncontextualized features or manual inspection. First, features extracted by related work are currently not robust enough to address this problem as they are easy to manipulate and bypass [15]. Second, human-dependent systems cannot keep up with the amount of malware being processed nowadays [16–18]. Moreover, the accuracy of manual inspection has been repeatedly questioned [19]. On the other hand, some systems are designed to extract a semantic core from a number of Android applications [20] or from their corresponding families [21, 22]. While promising, they suffer from a number of limitations, including scalability issues [20], memory complexity [21], and their excessive reliance on features extracted from specific code structures (e.g., loops) [22].

In this paper, we propose AndrEnsemble, an approach to characterize Android malware families. Unlike previous works, our method does not rely on a precise human-vetting process, which makes it more scalable and more suitable for learning in the presence of concept drift [23]. Also, instead of operating on a per-app granularity, it looks at groups of apps and leverages differential analysis to extract common family behavior [24]. Thus, AndrEnsemble

is more resilient to feature perturbations and manipulations. Our approach to build up a semantic-based core works as follows: we first create an aggregated call graph where each node represents a method and each edge shows an invocation between two different methods. Methods are represented as hashes computed with a custom fuzzy hashing function. Thus, similar methods across a group of samples are treated as a single node in the aggregated call graph. Aggregated call graphs have fewer nodes (since common hashes are considered as a single node) and connections as compared to individual call graphs. Both of these improve the performance of graph mining algorithms. Second, AndrEnsemble can deal with repackaged apps [25] more effectively. By looking at common methods in malware, non-popular methods (typically attributed to the original repackaged app) are discarded [24]. Additionally, a greedy algorithm is used to mine paths from the aggregated call graph depending on the maximum length which is justified. This makes the method applicable at different granularity levels similar to recent works [26, 27]. Each path may contain one or more calls to sensitive API methods. Therefore, less frequent edges in a family are also pruned to speed up the mining process.

Contributions. This work makes the following contributions:

- We propose a new characterization approach for Android malware families based on common ensembles of sensitive API calls. Contrarily to other related works which rely on individual API methods, we consider ensembles of API methods that are shared by a number of samples in each family. Also, our approach can be tuned to different granularity levels.
- We study and report common and rare ensembles of API methods in three types of Android malware: ransomware, SMS Trojans and banking Trojans. We discuss real examples for each type by linking these ensembles to apps' behavior.
- We report some anomalies that exist in the current family labeling of Android malware. In particular, we give examples of apps with identical and with very similar behavior despite belonging to different families.

2 APPROACH

This section presents our approach in detail. We first provide an overview of our system and then describe each component in detail.

2.1 System Overview

The system proposed in this work is composed of five main steps as depicted in Fig. 1. Given a specific Android malware family, it first computes the fuzzy hash values $(h_1, h_2, \ldots, h_n)$ of all extracted methods from applications $(a_1, a_2, \ldots, a_n)$ using a number of features $(f_1, f_2, \ldots, f_n)$. In parallel, we obtain the method call graphs of all apps $(g_1, g_2, \ldots, g_n)$. Next, the method call graph of each application is converted to a hash graph (*HG*), where nodes are hashes and edges are connections between hashes, all of which are reconstructed based on the methods connections of the method call graph. In the third step, hash graphs of different apps are all merged into an aggregated hash graph (*AHG*). We then extract paths using graph mining algorithms and we record ensembles of sensitive API methods $(s_1, s_2, \ldots, s_n)$ observed in these paths. Finally, app feature vectors $(v_1, v_2, \ldots, v_n)$ are created based on these ensembles. These vectors can be used as signatures to characterize family behavior.

2.2 Method Hashing (Step #1)

The first step involves computing fuzzy hash values of all methods extracted from the apps which do belong to the same family (see step #1 in Fig. 1). These values are used later to build call graphs and to extract sensitive API calls which are shared among a number of methods that resemble each other. This makes the system more resilient to transformation attacks compared to similar systems that rely on common sensitive API calls of identical method call graphs.

To generate a fuzzy hash value for each method (m_j) we consider several features, including the control flow graph signature created by Cesare's grammar (G_j) [28], a method's name (N_j) and its class name (C_j), a method's intents (I_j), its sensitive API calls (S_j), and, finally, native and incognito methods (M_j) found within the method. Thus, the method hashing process is performed by applying a regular hash function on these features extracted for each method [24, 25] by dividing them into pieces (or segments). These segments contain fragments of traditional hashes joined together for comparative purposes and are obtained using a rolling hash. A rolling hash makes use of a trigger value to determine the number of segments in each feature [29].

In what follows, we use the short form of h_j when we refer to the fuzzy hash value calculated for method j. A fuzzy hash value can be shared by two or more methods if they are exactly the same (*exact match*) or are slightly different (*approximate match*).

2.3 Building an Aggregated Hash Graph (Steps #2 & #3)

In the second step, we aim at creating a specific form of call graph, which we call aggregated hash graph (*AHG*), per family. Here, nodes are hashes obtained from step #1, and edges show whether or not there are connections between pairs of hashes (i.e., between different methods). An aggregated hash graph merges similar methods of different apps of a particular family into one node. This will thus speed up the extraction of common behaviors from the resulting graph.

To build this comprehensive graph, we first build a hash graph (*HG*) for each application separately. Thus, methods extracted from each app are assigned a hash value as described in Section 2.2. Then, hashes are connected to each other based on methods connections in the call graph (see #2 in Fig. 1). For instance, there is an edge between h_i and h_j $(h_i \rightarrow h_j)$ if and only if there is a call in method i to method j $(i \rightarrow j)$.

When such graph is generated for all apps in one family, an aggregated graph is obtained by simply merging common nodes (or hashes) and adding all edges that exist between each pair of nodes (see #3 in Fig. 1). Also, a unique weight $(w_1, w_2, \ldots, w_n)$ is assigned to each edge by summing up common ones in apps. The weight of an edge is 1 in the aggregated graph if it is present in only one application.

Therefore, an AHG is a weighted bi-directional graph for each family where each weight shows how many apps in the family share the same connection between two hashes and in the same direction. Thus, one can obtain meaningful insights about the common

Figure 1: Overall architecture of the proposed system.

Figure 2: A subgraph of the AHG extracted for the *oldboot* family with nodes and edges which are common in more than 70% of apps (Thickness of edges is a representative of their prevalence in the whole family).

behavior in a particular family by inspecting these connections and their corresponding weights. Fig. 2 shows a subgraph of the AHG extracted for the *oldboot* family with 11 apps. Here, each weight (represented by the thickness of an edge) shows how many apps share a particular method or an edge across the family.

The main difference between building an AHG and simply merging all call graphs of applications in a family is that, in the former case, similar methods are assigned with equal hashes and, thus, are considered as a single node. Instead, in the latter case, an indistinguishable change between methods renders different nodes. Therefore, our system is more resilient to automatic transformation attacks where we intend to extract common malicious behavior from a collection of apps belonging to a specific Android malware family.

2.4 Extracting Ensembles of API Calls (Step #4)

Building an aggregated hash graph per family using method hashes reveals two important pieces of information: i) the frequency of *similar methods* in all apps, ii) and the frequency of *methods calls* in all similar methods in a family. However, inspecting popular methods or calls gives a limited understanding of the behavioral capabilities of a malware family. Thus, in the last step, we extract ensembles of sensitive API calls observed either in methods or consecutive method calls of the aggregated graph (see #4 in Fig. 1).

API calls are appropriate representatives of an app's behavior. Sensitive API calls are those which can threaten user's security and privacy and can be used to perform various operations, ranging from OS-related to generic ones such as file system operations. We have considered 40 sensitive API calls which cover a wide range of activities as shown in [24]. These methods can be used for a variety of purposes. On the one hand, some API methods can be used to record device specific information (e.g., *DeviceId* or *SubscriberId*), location information, installed packages, and running processes or services. On the other hand, some of the API calls can be used to leak sensitive information to remote servers either by sending text messages or by establishing remote connections. Furthermore, we have considered methods by which malware specimens can manipulate critical information such as files (e.g., by creating, deleting or encrypting), processes and apps' contents that are handled by content providers. Finally, we have considered methods that are used by some apps to dynamically load classes and to deliver their malicious functionality at runtime.

It is worth noting that although some of these methods have been recently deprecated, only 20% of Android devices are running the newest major version of this OS [18]. This suggests that old Android malware with outdated API calls can still affect a large number of users, and, for this reason, we have not excluded these methods from our list. In addition, some of the methods considered here may not look sensitive alone but they could potentially be malicious when they appear with other API calls in an ensemble.

To extract ensembles of API calls from each family, we first identify all source methods in the AHG, i.e., those methods which are either isolated ($indegree = 0$ and $outdegree = 0$), or they have not been called from any other methods ($indegree = 0$). Afterwards, we extract all paths originating from source methods using a greedy path mining algorithm with respect to the weights of the edges.

This implies that all edges in a particular path do have a common frequency among apps which belong to the same family. Once these paths are extracted, we collect sensitive API calls appearing in each path. In particular, the union of sensitive calls along one path results in a unique ensemble of sensitive API calls for that specific path. So, we are finally left with several ensembles of API calls per family which have different percentages of prevalence among apps.

2.5 Creating Feature Vectors (Step #5)

Once ensembles of sensitive API calls are extracted, each application is assigned a binary feature vector. Each feature (named f_j in figures) is a unique ensemble of API calls in a family. The length of this vector for each app is thus equal to the total number of extracted ensembles from the entire dataset, and the presence or absence of an ensemble is shown with 1 or 0.

To measure similarities and differences between vectors, we use the cosine similarity metric as defined in Eq. 1. Specifically, given two vectors, A_1 and A_2, their cosine similarity is computed by scaling the dot product of these vectors to their magnitudes. Thus, the output is in the [0, 1] interval.

$$Sim_{cos}(A_1, A_2) = \frac{A_1 \cdot A_2}{\|A_1\|\|A_2\|} \quad (1)$$

Contrarily to the cosine similarity, the cosine distance expresses vectors dissimilarity in positive space (i.e., [0, 1]). This is done by subtracting the cosine similarity from 1 as follows: $Dist_{cos}(A_1, A_2) = 1 - Sim_{cos}(A_1, A_2)$. Thus, the cosine distance of two vectors is close to 0 when they are highly similar, and it is almost 1 when two vectors are completely different.

3 EVALUATION

In this section we evaluate our approach. We first present our experimental setting and describe our dataset. We then apply our system to about 700 families and 16K apps, and we present our results by grouping these families by type of family (i.e., ransomware, and two types of Trojans). We also describe how our approach can help in extracting the common malicious behavior of these families and how it finally leads to a fine-grained understanding of security sensitive operations which are exercised by each family.

3.1 Experimental Setting and Dataset

The proposed system has been implemented in Python. Our implementation extracts all method's features as well as call graphs using Androguard [30], a full Python reverse engineering tool developed for Android apps. We have evaluated our system on the biggest academic dataset of Android apps, known as AndroZoo [31]. Families have been all extracted using Euphony [11].

Experiments are all conducted on a 2.4 GHz Intel Xeon Ubuntu server with 40 CPUs and 128 GB of RAM. As we are interested in extracting common behavior from all apps in a medium and big size families, we discard those which contain less than 7 apps. Also, to alleviate the expensive process of path mining in large call graphs, we only extract paths with a maximum length of 2. Finally, we have excluded paths that include edges shared by less than 70% of apps. Therefore, if a family does not contain ensembles of sensitive API calls shared by more than 70% of apps, it is removed from the dataset as well.

AndroZoo contains around 8M Android apps from more than 3,000 families. The apps are gathered from 15 known markets and 1 unknown repository. However, the majority of them (≈ 97%) are collected from 3 main app markets, including Google Play, Anzhi and AppChina. Each app in this dataset is regularly scanned by various Anti-Virus (AV) vendors to separate malicious apps from benign ones. Around 1%, 33% and 17% of apps in the three markets in AndroZoo are malware according to at least 10 different AV vendors. Thus, our dataset of malicious apps is a subset of this huge market with 117 families and 3,050 malware specimens (see Table 1).

Table 1: Statistics of the dataset considered for case study, including the number of apps, number of families and the average size of applications (MB).

Malware Type	#Apps	#Families	Avg. Size
Ransomware	824	7	4.98
SMS Trojan	1,967	98	9.88
Banking Trojan	259	12	10.20
Total	3,050	117	8.35

3.2 Description of Experiments

Our system is evaluated with three types of malware families: ransomware (§3.3), SMS Trojan (§3.4) and Banking Trojan (§3.5) families. We have obtained the type of each family from AndroZoo. Our choice is motivated by the increasing popularity of these types of malware in recent years [17, 32].

For each type, i) we report the most common and rarest ensembles of sensitive API calls, ii) we present a case study to discuss one of the most popular families, and iii) we study the following two scenarios: a) where two apps from different families do share the same signature, and b) where two apps from different families have similar signatures (i.e., those that are different in two ensembles of API calls). These scenarios are used to provide an intra-family characterization. To report these scenarios, we rely on the cosine distance between the apps' feature vectors (as discussed in Section 2.5). All these three steps together allow us to confirm the applicability of our approach. We next describe each of the types of families studied. Furthermore, we have evaluated the time and memory complexity of our system for each type as summarized in Table 2.

Table 2: Average amount of time took in each step of our approach per family (in sec.).

Malware Type	AHG Extraction	Ensemble Extraction
Ransomware	35.51	266.17
SMS Trojan	44.92	44.31
Banking Trojan	57.19	67.81

3.3 Ransomware

Android ransomware families are categorized into two general groups, screen lockers and crypto ransomware [33, 34]. Apps from

the first group lock the smartphone screen, while those in the second group encrypt the victim's valuable files, both with the goal of extorting users to pay a ransom. Also, there are few families such as *Cokri* and *DoubleLocker* which have both capabilities. On the one hand, screen lockers follow three major strategies to achieve their goals, including activity hijacking, modifying specific parameters and disabling certain UI buttons. The main purpose of these strategies is to guarantee that the ransomware activity is always on top of other activities. On the other hand, crypto ransomware uses standard or customized crypto-systems to encrypt critical files.

Our dataset contains 824 ransomware samples from 7 different Android families as shown in Table 1. However, apps are not evenly distributed across families. For instance, the *svpeng* family has 604 specimens, whereas *jisut* contains only 4 malicious apps. In general, we could extract 25 ensembles of sensitive API methods by applying our method on ransomware apps. Experimental results (Fig. 3) show that 11 ensembles are present in more than 70% of ransomware specimens. Instead, only few ensembles are rare — they are present in less than 2% of apps (e.g., ensembles 4, 14 and 25 in Table 3). More details are presented in Appendix A.

3.4 SMS Trojan

From a general point of view, a Trojan is a type of malware that disguise itself as a legitimate application and commonly violates personal or confidential information stored on the device by performing secret operations. A smartphone Trojan can be seen as an application that affects the way a mobile device is being controlled [35]. Once installed on the victim's device, it performs a wide range of silent activities, ranging from harvesting user or device specific information to intercepting incoming and/or outgoing text messages, sending premium SMS messages and connecting the device to a botnet to name a few. As Android Trojans commonly masquerade as popular legitimate apps available in official markets, they affect a large number of users.

SMS Trojans are malware specimens that usually monetize users by sending text messages to premium rate numbers [9, 36]. Our dataset contains 1,967 apps from 98 SMS Trojan families. We have extracted 168 different ensembles of API methods from these apps. Results show that 3 ensembles of API methods (i.e., *<delete(), exists()>* and *delete()* and *getClassLoader()*) are present in more than 50% of apps in the different families. On the other hand, almost half of the ensembles are specific to very few apps in our dataset. In particular, 91 ensembles of API methods (54%) are present in less than 2% of apps in the SMS Trojan families. Also, ensembles with length 2 are more prevalent among SMS Trojan families as compared to ransomware. In addition, two long ensembles with 6 sensitive API methods exist in 5 malware specimens. More details are provided in Appendix B.

3.5 Banking Trojan

The main goal of banking Trojans is to steal banking or credential information. They usually do this by either intercepting SMS messages [37], or by overlaying a fake window on top of other financial apps and websites [38]. In addition, other variants of Android banking Trojans may have some additional capabilities. Studies show that most of banking Trojans target specific geographical locations. For example, Russia and Australia are usually on top of this list [37].

Our dataset contains 259 apps and 12 Banking Trojan families from which we have extracted 50 unique ensembles of sensitive API methods. There are 2 ensembles of API methods, including *getClassLoader()* and *getInputStream()* that are shared by more than 50% of apps from different families. This means that more than half of apps in different families intercept from open connections, and they load their malicious classes at runtime like SMS Trojan applications and similar to the very recently detected variant of Rotexy family[1]. On the contrary, there are 9 API ensembles which are common among less than 5% of apps. Also, ensembles of length one are more prevalent among banking Trojans than ensembles of other lengths. More details are presented in Appendix C.

4 CONCLUSION AND FUTURE WORK

In this paper, we proposed a new approach to characterize Android malware families based on ensembles of API methods that are exercised by the majority of apps. Instead of relying on individual API calls, we extract ensembles of API calls to make the approach more resilient against transformation attacks. In addition, API ensembles provide the analysts with more meaningful insights of the behavior of an app. We make use of a fast graph-mining algorithm to extract these common and sensitive API ensembles from an aggregated form of method call graph.

Experimental results obtained from applying our method to three types of Android malware, including Ransomware, SMS Trojans, and banking Trojans reveal several interesting findings. First, malicious operations do not necessarily contain several sensitive API methods. In fact, a considerable number of common ensembles ($\approx$ 72% in ransomware, $\approx$ 21% in SMS Trojans, and $\approx$ 52% in banking Trojans) contain only one sensitive API method. Second, opposite to ransomware and banking Trojans, ensembles of two API methods were the most common in SMS Trojans. Finally, we found several samples with identical ensembles though belonging to different families.

This work can be extended in various ways as future work. More exhaustive static analysis tools such as Soot[2] can be used to extract call graphs. Additionally, a query-like based system can be leveraged to mine a dataset for threat discovery. Also, ensembles of API calls could be mapped to relevant behavior by developing and training an expert system.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We are also grateful to Yanick Fratantonio for his feedback on an early draft of this paper. This work has been supported by the Comunidad de Madrid (Spain) under the grant CYNAMON (P2018/TCS-4566), co-financed by European Structural Funds (ESF and FEDER). Also, it has been partially supported by the EPSRC under grants N028112 and N008448.

[1] https://securelist.com/the-rotexy-mobile-trojan-banker-and-ransomware/88893/
[2] https://sable.github.io/soot

REFERENCES

[1] Statcounter. 2018. Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share.

[2] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. 2018. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *Proceedings of the Internet Measurement Conference 2018*. ACM, 293–307.

[3] Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. 2018. Honey, I Shrunk Your App Security: The State of Android App Hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 69–91.

[4] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. 2018. FlowCog: context-aware semantics extraction and analysis of information flow leaks in android apps. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1669–1685.

[5] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, Xiaofeng Wang, Long Lu, and Haixin Duan. 2018. An empirical study of web resource manipulation in real-world mobile applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1183–1198.

[6] Omid Mirzaei, Jose M. de Fuentes, Juan Tapiador, and Lorena Gonzalez-Manzano. 2019. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* 90 (2019), 240–261.

[7] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. 2018. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*. 18–21.

[8] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. 2014. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications* 41, 4 (2014), 1104–1117.

[9] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 329–334.

[10] Pavel Laskov et al. 2014. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 197–211.

[11] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 425–435.

[12] Sevil Sen, Emre Aydogan, and Ahmet I Aysan. 2018. Coevolution of Mobile Malware and Anti-Malware. *IEEE Transactions on Information Forensics and Security* 13, 10 (2018), 2563–2574.

[13] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, Bonn, Germany, 252–276.

[14] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 230–253.

[15] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

[16] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. 2016. Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 122–141.

[17] Kaspersky Lab. 2018. Kaspersky Lab Threat Predictions For 2018. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07164714/KSB_Predictions_2018_eng.pdf.

[18] Symantec. 2018. Executive Summary - 2018 Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-executive-summary-en.pdf.

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

[20] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security Conference*. Springer, 513–527.

[21] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[22] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2018. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 112–123.

[23] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *PROCEEDINGS OF THE 26TH USENIX SECURITY SYMPOSIUM (USENIX SECURITY'17)*. USENIX Association, 625–642.

[24] Guillermo Suarez-Tangil and Gianluca Stringhini. 2018. Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned. *arXiv preprint arXiv:1801.08115* (2018).

[25] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 317–326.

[26] Fady Copty, Matan Danos, Orit Edelstein, Cindy Eisner, Dov Murik, and Benjamin Zeltser. 2018. Accurate Malware Detection by Extreme Abstraction. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 101–111.

[27] Omid Mirzaei, Guillermo Suarez-Tangil, Juan Tapiador, and Jose M de Fuentes. 2017. Triflow: Triaging android applications using speculative information flows. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 640–651.

[28] Silvio Cesare and Yang Xiang. 2010. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*. Australian Computer Society, Inc., 61–70.

[29] Dustin Hurlbut-AccessData. 2009. Fuzzy Hashing for Digital Forensic Investigators. (2009).

[30] Geoffroy Gueguen. 2012. Androguard. https://github.com/androguard/androguard.

[31] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 468–471.

[32] Christiaan Beek, Diwakar Dinkar, Yashashree Gund, German Lancioni, Niamh Minihane, Francisca Moreno, Eric Peterson, Thomas Roccia, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, RaviKant Tiwari, and Vincent Weafer. 2017. *McAfee Labs Threats Report*. Technical Report. McAfee Labs.

[33] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. 2018. Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE Transactions on Information Forensics and Security* 13, 5 (2018), 1286–1300.

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

[35] Mikko Hyppönen and Tomi Tuominen. 2017. F-Secure State of cyber security. https://www.f-secure.com/documents/996508/1030743/cyber-security-report-2017.

[36] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 25. 50–52.

[37] Roman Unuchek. 2017. A new era in mobile banking Trojans. https://securelist.com/a-new-era-in-mobile-banking-trojans/79198/.

[38] Lukas STEFANKO. 2018. Banking Trojan found on Google Play stole 10,000 Euros from victims. https://lukasstefanko.com/2018/09/banking-trojan-found-on-google-play-stole-10000-euros-from-victims.html.

[39] Lukas Stefanko. 2015. Aggressive Android ransomware spreading in the USA. https://www.welivesecurity.com/2015/09/10/aggressive-android-ransomware-spreading-in-the-usa.

[40] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. 2017. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security* 12, 8 (2017), 1772–1785.

[41] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76.

APPENDIX

A RANSOMWARE

Case Studies. To confirm the validity of our approach, we select the most recently detected ransomware families from our dataset and discuss their app feature vectors in detail. Also, we compare

Figure 3: Distribution of sensitive API call ensembles among ransomware samples from different families.

the results with what is known from each family from other research works and security reports. We have selected the *porndroid* and *gepew* ransomware families both of which with apps that are present in the wild since 2014.

Table 3: Mapping between feature numbers and ensembles of sensitive API methods extracted from ransomware families.

#Feature	Ensemble of Sensitive API Methods
1	delete(), exists()
2	getClassLoader()
3	insert(), query()
4	insert()
5	openConnection()
6	connect()
7	getInputStream()
8	getApplicationInfo()
9	getSubscriberId()
10	openConnection(), connect(), getInputStream()
11	getFilesDir()
12	mkdir(), exists()
13	crypto
14	loadLibrary()
15	query()
16	mkdir()
17	setDataAndType()
18	exists(), mkdir()
19	exists()
20	delete()
21	getDeviceId()
22	getActiveNetworkInfo()
23	mkdir(), delete(), exists()
24	addFlags()
25	openConnection(), connect(), getInputStream(), getFilesDir(), exists()

Porndroid is a ransomware family which hides behind a fake pornography app. Security reports show that once an app from this family is downloaded, it downloads another file, known as LockerPin [39]. Then, the user is locked out of the device and the pin number of the phone is changed. The apps in this family usually display a warning window from an official source (e.g., security agencies like FBI) and threaten the victims to pay the ransom in return for the illegal pornographic websites they have accessed with their smartphones. If this ransom is not paid, all files are deleted and the phone is restarted to its factory settings.

Our dataset contains 10 different apps from the *porndroid* family and 8 ensembles of API calls are shared by more than 70% of apps. The behaviors shown by these ensembles of APIs are aligned with actions described by commercial reports and threat intelligence gathered from each of the families. In particular, all samples (100%) do contain 3 common API calls, including *getActiveNetworkInfo()* and *getClassLoader()* and *addFlags()*. The first method is used to obtain details about the current active data network and can be used to check whether or not the compromised device is connected to the Internet. Once confirmed, the apps download the original malicious application (i.e., LockerPin). The second method is used to retrieve the loader of a specific class at runtime. This implies that all apps in this family share malicious classes which are not installed as part of the application package and are loaded dynamically during execution. In other words, all apps execute the LockerPin application once it is downloaded successfully. Finally, the third method is used to make sure that ransomware activity is overlaid always on top of other activities. This is done to prevent the victim from accessing other components of the device. The ransomware can set the value of this method to "FLAG_ACTIVITY_NEW_TASK" to restart itself and overwrite previous activities whenever the ransomware is not displayed on top.

Moreover, 90% and 80% of apps in this family include *query()* and *delete()* API methods respectively. These are present in 3 ensembles as shown in Table 3. The former method is used to retrieve and leak victim's personal information through Android content provider [40], while the latter is used to delete critical files should the ransom not satisfied.

Intra-family characterization. The last step in our evaluation is to look at the intra-family dependencies. For this, we compare the feature vector of an app with all other apps by using cosine distance. The results are presented in Fig. 4a. Our method reveals several cases where apps in two different ransomware families have exactly the same signature. For instance, *4654EC...48F2.apk* from *slocker* family and *8905B3...99DC.apk* from *gepew* family do share exactly the same feature vector. Further inspections show that both of these apps do contain methods which are not installed as parts of both apps' packages. Thus, they make use of dynamic loading to retrieve the class loader of those methods and to load the malicious methods into memory at run-time (feature #2 in Table 3). This comes to show the effectiveness of our approach to understand common behaviors between two families with the same capabilities.

We also look at the dual: when apps of two different families have different feature vectors. For instance, *C3829A...03DB.apk* from *svpeng* family and *877D3B...2AE4.apk* from *slocker* family are different in two features. In particular, the first app overlays its window on top of other windows (feature #24 in Table 3). This app also has a keyword database to identify encryption-related words in UI widgets (feature #15 in Table 3). This is similar to variants of the *ransomprober* family [33]. Instead, the the second app does not have any of these capabilities. These two differences are the main distinctions between the two families. However, all other features are shared. This indicates that two families have evolved from one another. It also shows to what extent we can use our system to explain the differences between apps.

B SMS TROJAN

Case Studies. Like in the previous section, we select two of the most popular families (i.e., *Cvmtld* and *Rusms*) and provide a qualitative evaluation or our findings.

(a) Ransomware **(b) SMS Trojan** **(c) Banking Trojan**

Figure 4: The cosine distance of app vectors in different families for each type of malware (values close to 0 show high similarity, whereas those close to 1 show significant difference). Two or more groups of apps with colors close to blue in one row are those which are behaviorally similar in different families.

Cvmtld is an Android SMS Trojan which contains 19 different samples out of which: one sample is first detected in 2013, five samples are first detected in 2014, and the rest have been all detected in 2016[3]. We observe that 8 ensembles of API methods are shared among all samples in this family. However, if we look at lower granularity levels, we observe informative ensembles as well. All specimens collect sensitive device information and leak them to remote servers. They are also able to detect emulators and to evade dynamic analysis (*<myPid(), killProcess()>*) similar to other families reported in earlier works [41]. As these behaviors are common to all samples in the family, we can say that these are the "core" capabilities that characterize the family. However, there are other sets of behaviors that can be used to characterized variants of the family. As it is clear, 95% of samples send text messages to specific numbers (*sendTextMessage()*). There are also behaviors that enable these apps to update their capabilities during runtime (*getClassLoader()*).

Intra-family characterization. There are several examples (see Fig. 4b) where two apps from different SMS Trojan families have exactly the same behavior. *2F7794...88B8.apk* from *smsboxer* family and *971913...5B82.apk* from *darrma* family share exactly the same feature vector although they belong to different families. Both of these apps are equipped with mechanisms to detect dynamic analysis systems and can halt their activities if they observe clues of simulated environments. Also, they rely on Internet connection to deliver their malicious functionalities and can encrypt/delete files.

When looking at apps with different behaviors, we observe *CD3A15...768D.apk* from *moavt* family and *229C9A...8357.apk* from *darrny*. The first app collects the phone number string (via *getLine1Number()*) whereas the second one does not. In contrast, the second app is able to detect emulators (using *<myPid(), killProcess()>* ensemble of API calls) as compared to the first app.

C BANKING TROJAN

Case Studies. We next describe one of the most prevalent banking Trojans according to our dataset, known as *Fareac*.

[3]https://www.virustotal.com

Fareac is a banking Trojan family which contains 37 different malware samples in our dataset. The results obtained reveal 27 different ensembles of API methods which are shared by all applications in this family. In particular, all apps in this family have similar behavior and share common characteristics. First of all, they check whether or not the WiFi connection is enabled on the target device (*isWifiEnabled()*). Once this is clarified, they load a native library into memory (*loadLibrary()*) and call the loader to execute all loaded malicious classes (*getClassLoader()*). Then, they steal victim's credential information (using ensembles such as *<setFlags(), getApplicationInfo()>* and *query()*) and some extra information (e.g., network operator using *getNetworkOperator()*) by overlaying the windows of other legitimate apps and services (*addFlags()*). Once these information are gathered, they encrypt all of them (*crypto*) and leak them to remote servers by opening a connection (*<openConnection(), connect()>*). Also, they delete original files after encryption procedure (using *exists()* and *delete()*).

Apps in *Fareac* family are also able to intercept data from current open connection (*getInputStream()*). Additionally, they can all detect simulated environments (*<killProcess(), myPid()>*), and, thus, can bypass dynamic analysis. Indeed, this family is very hard to be detected as it can potentially evade both static and dynamic analyses.

Intra-family characterization. Similar to ransomware and SMS Trojans, we have inspected several cases (see Fig. 4c) of two apps in different banking Trojan families with identical and different behavior. For instance, *6B03C9...807C.apk* from *sodsack* family and *3DC0F8...D204.apk* from *ztorg* family are two bankers which load their malicious code at run-time to evade static analysis. On the other hand, *3CC01D...7012.apk* from *dhyvax* family and *748ECD...0837.apk* from the *acecard* family have different capabilities. The former banker checks the list of files in different directories, and creates files and folders, whereas the latter is able to delete files or directories.