Compact Explanations of Why Malware is Bad

Compact Explanations of Why Malware is Bad

(Extended Abstract)

Wei Chen[1], Charles Sutton[1], Andrew D. Gordon[1,2], David Aspinall[1], Igor Muttik[3], and Qi Shen[4]

[1]University of Edinburgh, UK
[2]Microsoft Research Cambridge, UK
[3]Intel Security, UK
[4]Peking University, China

Researchers and malware analysts have identified hundreds and thousands of mobile applications as malware. These malware instances are organised into families based on some common unexpected behaviours, e.g., send premium messages, access locations, and intercept incoming messages and calls, etc. However, except some unclear online technical descriptions of several famous malware families, to the best of our knowledge, people have no idea of what exactly happens in mobile malware or what kind of behaviour of a mobile application makes it bad. This brings a challenging research problem: to automatically generate compact and precise explanations of unexpected behaviours in a mobile application if it has been identified as malware.

This research has several potential benefits, including: help people get better understanding of potential threats hidden in mobile applications; provide hints for malware analysts before more expensive investigation; support automatic generation of malware analysis reports; and provide clear and friendly references for security policy designers, etc.

Some fundamental technical questions we will answer are as follows. How could we characterise and formalise an application's behaviour as efficiently and precisely as possible? What kind of behaviour is unexpected with respect to a specific application and how to figure it out automatically? Once a certain behaviour has been identified as unexpected, how could we automatically generate an explanation of this behaviour and in what kind of form? Finally, how could we evaluate generated explanations?

To answer these questions we develop the following approach.

Behaviour Abstract. We formalise an application's behaviour as an automaton, which is actually an extension of a call-graph: valid control-dependency sequences of events, actions, and annotated API calls allowed by an application, so-called a behaviour automaton. The choice of which features to include is a trade-off between efficiency and precision. Automata are not as precise as data-flows but much more accurate than manifest information, e.g., permissions and actions, and API calls appearing in code which might contain "noise" caused by dead code and libraries. Also, compared with APIs, automata can capture more sophisticated behaviours. This is needed since some unexpected behaviours only arise when APIs are called in certain orders. On the other hand, it is much easier to generate behaviour automata for applications en masse than data-flows, in particular, people can annotate appealing APIs to generate behaviour automata more efficiently, instead of considering all data-dependency between statements. We have designed and implemented a static analysis tool to construct a behaviour automaton directly from each Android application to approximate its behaviour; to develop such a tool, we have to consider a broad range of features of the Java and the Android

framework, e.g., multi-threads, multi-entries, component life-cycle, inter-component communications, and runtime-registered listeners, etc.

Feature Construction. Fixing a group of applications which have been decided as malware or benign, once a behaviour automaton has been constructed for each of them, we want to figure out which behaviour (a sub-automaton) is unexpected and which is normal by applying machine learning methods. But, before any interesting pattern exploration, we have to decide which part of an application's behaviour is exclusive to itself and which part is shared with other applications; this we call feature construction. Since the space of features, which is consisted of intersection and difference between automata, in theory, is at exponential order of the number of sample applications, we approximate this space by searching for a salient subspace. This search process is guided by behavioural difference between malware and benign applications. We combinatorially construct features from a group of malware instances by doing intersection and difference between their automata; to narrow the searching space we get rid of all non-salient features constructed in previous step, here, a feature is salient if is actually used in a classifier, which is trained from these malware instances mixed with an equal number of randomly-chosen benign applications; we combine features by doing intersection and difference between features from different groups; we design a divide-and-conquer-based multi-processing algorithm to construct and combine features in the above way more efficiently; this process stops until all sample malware instances have been considered.

Learning Unexpected Behaviour. An unexpected behaviour could be a common behaviour shared by malware instances and rarely seen in benign applications, e.g., intercept incoming messages and send them out via Internet connections, load classes from a hidden payload and execute commands from remote servers, and send premium messages constantly, etc. An unexpected behaviour in a group of applications could be normal in another, e.g., accessing locations is common in Jogging Tracer applications but unexpected in E-Reader applications, sending messages is the basic functionality of messenger applications but unexpected in Card Game applications, and accessing Internet can be seen in a lot of applications but unexpected for a FlashLight application, etc. These observations lead us to two methods of learning unexpected behaviours, i.e., to learn prevalent common patterns from malware families; and to identify singular behaviours of a group of malware and benign applications whose behaviours are very similar. For the first method, for each malware family, we search for a subset of all its behaviours, which maximizes the probability of a malware instance belonging to this family, if this malware instance has all behaviours in this set; we take the union of these sets as unexpected behaviours learned from malware families. As for the second method, we group applications by applying clustering methods using normalised compression distances between regular expressions of their automata, so-called context; for each group of applications, we train a classifier and choose features as unexpected behaviours by weight ranking. There are definitely some unexpected behaviours that cannot be captured using the above methods, e.g., accessing Internet is a behaviour of almost every FlashLight application although it is unexpected with respect to these applications. This case is out of range of our consideration. Except explanation learned unexpected behaviours will be used in malware detection as well: a target application is decided as malware if it has any unexpected behaviour from malware families or the classifier of its context consider it as malware.

Generating Explanation. We want to explain identified unexpected behaviours to a broad range of people, including: general users, malware analysts, security experts, policy designers, and researchers, etc. To satisfy different requirements for information accuracy and technical details, we produce explanations in three forms: short paragraphs, abstract and concrete behaviour automata, and statistical charts. A short paragraph is to give people rough ideas of what might happen in a malware instance, i.e., from unexpected behaviour automata, we extract actions, events, APIs, and sub-sequences of them; and feed their names as keywords

through pre-defined templates to generate paragraphs, or use these keywords to search for sentences from manually-produced malware analysis reports as explanations. This form of explanation is readable and brief but less informative or precise for malware analysts, security experts, and researchers, etc. We present on-demand to these people behaviour automata or abstract automata in which API names are replaced by permission-like phrases. Additionally, we generate prevalence pie charts and prevalence-time-series charts to show the changes of unexpected behaviours in malware samples over a long period, so as to help reveal the evolution of unexpected behaviours. Here are explanations for instances from malware families. They are generated by sentence searching.

§ *This is a trojan which steals personal information from the infected device. It can be controlled over the web through http. (DroidKungfu)*

§ *It sends sms messages to premium rated numbers. (OpFake)*

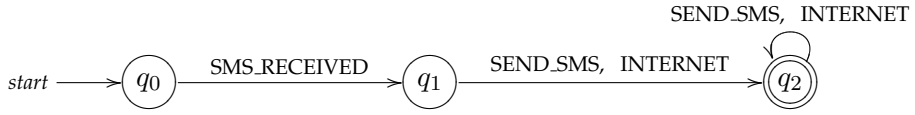
§ *Allows applications to open network sockets, and uploads the data to a specific url. (Zitmo)*

Some explanations generated from pair features which are learnt in application context are as follows.

§ *It is declared to be a Chatting application, but, after a USB massive storage is connected, it will: retrieve a class in a runnable package; read information about networks; connect to Internet. (BaseBridge)*

§ *It is declared to be an Anti-Virus application, but it will: read your phone state after a phone call is made; read your phone state then connect to Internet; send SMS after a phone call is made. (Zitmo)*

As a demonstration of more precise explanation, an example of behaviour automata is given as follows.



It actually captures the unexpected behaviours of malware family Zitmo.

Evaluation. We evaluate the quality of automatic explanation in three ways. First, by using unexpected behaviours as input features, we achieve a malware detector with good classification performance. Second, we do a user-study based on surveying randomly chosen people. The survey compares our approach with several alternative methods. Participants are invited to choose explanations they prefer. They are also asked to score each explanation to indicate to what extent it is convincing. Our results show automatic explanation can improve people's belief on the system's automatic decision. Third, we collect human-authored descriptions of identified malware families in online malware analysis reports from different sources. A subjective comparison shows automatic explanations are comparable to these manual descriptions.

We have presented a new approach to automatically generate explanations of unexpected behaviours for mobile applications, provided they have been automatically classified as malware. In contrast, most previous work on malware detection focused on obtaining good fits to a given training set by trying different methods and variant kinds of features [1, 3, 4, 10]. Explanations of chosen features produced by machine learning have received much less consideration. To the best of our knowledge, Drebin [2] is the first attempt to automatically generate explanations of Android malware. Drebin generates an explanation by choosing syntax-based features with top weights from a classifier and processing these features through templates to output text. But, syntax-based features cannot capture sophisticated behaviours. We overcome this limitation by adopting semantics-based features, i.e., behaviour automata.

The idea of using extended call-graphs to capture application behaviours is close to research by Chen et al. [5] and Yang et al. [9]. The former proposed permission-event graphs to capture

application behaviours for verification. The latter created two-tiered behavioural graph model to capture malware program logic for malware detection. Call-graphs were also exploited to help malware detection [7]. Among others, the method Dendroid [8] used cosine similarity between vectors of call graphs of identified malware to classify unknown malware samples into identified families. Similar ideas were applied in DroidLegacy [6] to detect piggybacking by exploiting difference between API sets of modules of malware in different families. All of these tools and methods were developed to eliminate unexpected behaviours without further explanations of why some behaviour is bad.

In future work, we want to construct anti-security policies from unexpected behaviours to help application verification and zero-day malware detection. We will focus on how to use verification results as semantics-based features to improve classification performance for zero-day malware detection.

References

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In T. Zia, A. Y. Zomaya, V. Varadharajan, and Z. M. Mao, editors, *SecureComm*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 86–103. Springer, 2013.

[2] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. *Proc. of 17th Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[3] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2010.

[4] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[5] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[6] L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 3:1–3:12, New York, NY, USA, 2014. ACM.

[7] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54, New York, NY, USA, 2013. ACM.

[8] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.

[9] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, September 2014.

[10] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new Android malware detection approach using bayesian classification. In L. Barolli, F. Xhafa, M. Takizawa, T. Enokido, and H.-H. Hsu, editors, *AINA*, pages 121–128. IEEE Computer Society, 2013.