NTApps: A Network Traffic Analyzer of Android Applications

Rodney Rodriguez, Shaikh Mostafa, and Xiaoyin Wang
Department of Computer Science, University of Texas, San Antonio
One UTSA Circle, San Antonio, Texas, 78249
{rodney.rodriguez, shaikh.mostafa, xiaoyin.wang}@utsa.edu

ABSTRACT

Application-level network-traffic classification is important for many security-related tasks in network management. With the knowledge of which application certain network traffic belongs to, the network managers are able to allow/block certain applications in the network (whitelisting/blacklisting), or to locate known malicious applications in the network. To support application-level network-traffic classification, the network managers require a network-signature for each possible applications in the network, so that they can match these signatures with the network traffic at runtime to identify the ownership of the traffic. The traditional approaches to generating network-signatures for applications require either manual inspection of the application or accumulated annotated network traffic of the application. These approaches are not efficient enough nowadays, given the recent emergence of mobile application markets, where hundreds to thousands of mobile apps are added everyday. In this paper, we present a fully automatic tool called NTApps to generate network signatures for the mobile apps in android market. NTApps is based on string analysis, and generates network signatures by statically estimating the possible values of network API arguments.

ACM Reference format:
Rodney Rodriguez, Shaikh Mostafa, and Xiaoyin Wang. 2017. NTApps: A Network Traffic Analyzer of Android Applications. In *Proceedings of SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA, ,* 8 pages.
DOI: http://dx.doi.org/10.1145/3078861.3084175

1 INTRODUCTION

Application-level network-traffic classification is a preliminary requirement for many security-related network management tasks. For example, after classifying network traffic to applications, the network managers are able to enforce white list or black list of applications to block certain network-using applications. Furthermore, based on the classification results, the network managers are also able to check whether known malicious applications are used by some terminals in the network, and further locate which terminals are running the malicious application. Moreover, the network managers may also use the network-traffic classification tool to analyze logged network traffic history to discover suspicious human behaviors. The problem of automatically identifying the applications generating network traffic is called *application identification*. A *traffic classifier* is a tool for application identification. Payload-based classifiers inspect and compare the contents of payloads with a pre-existing signature database. Payload-based techniques have higher accuracy than other classifiers [33] and have been used as ground-truth in comparative studies of traffic classifiers [19].

To perform application-level network-traffic classification, the current practice in the industry is to generate a network signature for each application so that the network management system can match network traffic with these signatures to decide which applications the traffic may belong to. The signatures can be based on network-traffic features (throughput, intervals, etc.) [24] [13] or based on the content of the packets in the traffic [34] [15]. The traffic-feature-based signatures are often not precise enough, because different applications may share similar network-traffic features and network-traffic features may be affected by various environment factors (e.g., the network speed, number of terminals). By contrast, the content-based signatures are much more precise and robust, and therefore are especially suitable for security-related network-management tasks. Though initial payload techniques could not handle encrypted traffic, later work showed that encrypted traffic can be handled using distributions over payload contents [22]. Generating a signature database for payload-based classification is time consuming, requiring a large volume of network traces and manual effort for signature construction.

However, to generate content-based network signatures for all possible applications in the network is far more than trivial work. The existing approaches to generating these signatures fall into two categories. The first category of approaches[34] depend on manually inspecting the code or generated network traffic of an application, which is tedious and costly. The second category of approaches [21] [27] try to automatically extract network signatures of an application from a large amount of its network traffic, using data mining techniques. Although the extraction phase is automatic, the accumulation of network traffic usually takes non-trivial time.

Recently, the emergence of mobile application markets brings new challenges to the above existing approaches. Both Apple App Store[1] and Google Play Store[2] hold hundreds of thousand apps, and several hundred apps are added to the markets each day. The low efficiency of the existing approaches makes them difficult to keep up with the growing speed of the number of apps in the mobile application markets. Therefore, more efficient and automatic approaches are of eager requirement. In this paper, we propose a novel fully-automatic approach to generate network signatures for large numbers of android apps. The basic idea of our approach is

SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/3078861.3084175

[1] http://store.apple.com/
[2] https://play.google.com/

to statically analyzes the byte code of an android app, and to use string analysis to estimate the possible contents of the packets sent to the network[3]. Our approach is based on the observation that, the content of the sent network packets are usually generated with one or more packet-generation APIs (which we refer as network APIs in the rest of this paper) by concatenating the arguments of these APIs. Therefore, we will be able to estimate the content of sent network packets, if we are able to estimate the possible values of the arguments of network APIs, and to modelling the network APIs on how they generate the contents of network packets. We use program analysis to avoid the time-consuming steps of data collection and manual signature construction. The modular nature of program analysis allows us to tune the precision of signatures generated by use of different abstractions. Though our framework is general, we follow the practice in the literature, by instantiating and evaluating our work on Android applications. In the literature, the word application is sometimes used for application layer protocols. In this paper, the words application and app always refer to smartphone applications.

In particular, our approach consists of the following five steps. First, for a given android app, we translate the Dalvik byte code in its apk file to Java byte code using our existing tool. Second, we locate in the Java byte code all the invocations of the network APIs that are in our pre-defined network API list[4]. Furthermore, for each network API invocation, we build an *API grammar summary* which presents how the API manipulates its arguments to generate a network packet. Third, we set each argument *arg* of these located network API invocations as the input of string analysis and perform string analysis on the byte code to generate a string-operation grammar that estimates the possible values of *arg*. Fourth, for each network API invocation, we combine its API grammar summary with the string-operation grammars of all of its arguments, to generate a *combined grammar* that is able to estimate the network-packet content generated by this network API invocation. Fifth, we extract network signatures as a set of constant-string sequences, from all the combined grammars of an app.

Our work builds upon insights about traffic generated by smartphones and the structure of smartphone applications. The first insight is that a significant portion of smartphone traffic is HTTP. A recent study found that between 92% to 97% of traffic generated by handheld devices is unencrypted HTTP [11]. In contrast, 72% to 81% of traffic generated by larger devices is HTTP. The same study also found that 82% of the HTTP traffic consumed by smartphones is related to non-browser applications, while only 10% of the HTTP traffic of larger devices is not browser related. Thus, traffic classification based on payloads is feasible, and will provide insight into the applications operating on a network.

Our second insight is that smartphone applications are downloaded and installed from a small number of application markets. Download and installation statistics available from application markets provide data about applications operating on a network. Studies have shown that the applications used on a network vary greatly depending on the time of day, week, and geographical location [49]. Collecting a representative sample of software run by users is possible and requires significantly less time and effort than collecting a representative sample of traffic generated by users.

The third insight is that smartphone applications use a small set of APIs to generate network traffic. A network signature for an application, or a class of applications is a summary of the data that is passed as an argument network APIs. Viewed this way, network signature generation can be reduced to a static analysis problem.

Static analysis has several advantages, which we describe briefly below and justify empirically in the paper. The main advantage is automation. Signatures need not be constructed manually or generated from network data, both of which are time consuming. The second advantage is flexibility in choosing the trade-off between precision and efficiency. The quality and efficiency of signature generation can be tuned by plugging abstractions of different precision into a static analyzer. Constant propagation with strings is extremely fast and produces coarse signatures. Inter-procedural analysis with context free grammars produces detailed signatures but is more expensive. From an implementation perspective, the lattice-based static analysis framework allows us to easily implement different signature generation techniques. Different types of signatures can be generated by changing a single component of the classification infrastructure, rather than changing the entire infrastructure.

The goal of our analysis is to summarise the data sent over a network. Like much existing work on smartphone applications, we focus on Android applications. Unlike existing static analyses for Android [8, 10], which focus on control rather than data-flow, we need to summarise the contents of payloads in an application. Designing such an analysis involves several challenges that have not been addressed by existing work.

The rest of this paper is organized as below. Section **??** presents the major challenge in our tool construction. Section 3 presents the overview and components of our tool architecture. Section 4 presents how our tool can be started and its input / output. Then we discuss some important issues in Section 5. Before we conclude in Section 7, we introduce a list of related research efforts in Section 6.

2 CHALLENGES OF TOOL CONSTRUCTION

The first significant challenge is the event-driven nature of Android applications. Program analysis is based on computing fixed points using control-flow or data-flow graphs. The flow of control in Android applications is largely determined by system callbacks, so there is no explicit flow of control to the entry point of methods. Event handlers must be considered by our analysis because the network is often accessed in response to user events. To discover which methods are called, the static analyzer must be aware of application lifecycles, listeners, and callbacks to event handlers.

The second challenge is to model APIs used to construct and manipulate payloads. Android applications extensively use `java.lang.string`, `java.lang.URL`, `java.lang.URLConnection`, `org.apache.http`, among other APIs to access and process data that is sent over the network. The analysis must precisely capture the semantics of such APIs.

The third challenge is inter-procedural analysis. Existing analyses of Android applications are either intra-procedural [8], or follow

[3]It should be noted that although the detailed design and implementation of our approach is for android apps, the basic idea of our approach may be applicable to other mobile apps or even PC applications

[4]Note that this list is generated manually only once and then used when generating network signatures for all android apps.

method calls in a limited way [10], or do not track data [12]. Data that is sent over the network is usually constructed and sanitized using several different methods in an application. In our experiments, we have found that inter-procedural analysis is sometimes required to even derive the hostnames an app interacts with. Up to 61% of applications in some data sets have been observed to use reflection [10]. Our analysis must also handle reflection to produce useful results. The problem we address is automatic generation of network signatures for Android applications.

3 TOOL STRUCTURE

As mentioned above, network signatures of mobile apps are very useful for packet inspection and network management. Figure 2 shows how our NTApps tool can be integrated with network management tools. Specifically, after NTApps generates signatures for various apps, the signatures as a tree or automaton can be fed into the packet classifier in the gateway of the network. NTApps is based on static analyzing the apk files of android apps. The overview of our approach is presented in Figure 1. From the figure, we can see that the input of our approach is an apk file and a pre-generated list of network API specifications. The output of our approach is a network signature, which is in the form a set of constant sequences which can be presented as a tree by combining common prefixes.

The NTApps tool includes five main components. The first component uses the our existing tool to convert the Dalvik code in the apk file to Java byte code. The second component is for network API handling. It locates all the network API invocations in the Java byte code, and will further build an API grammar summary for each the network API invocations according to the semantic of the API, based a list of pre-generated API grammar templates for network APIs in the android library. The third component uses the arguments of the located network API invocations as input, and apply string analysis to the Java byte code, so that this component is able to generate an context-free grammar for each argument. The fourth component of NTApps combines the API grammar summary *sum* of each network API invocation *inv*, with the context-free grammar generated by the string-analysis component for each argument of *inv*, and form a combined grammar of *inv*. The last component takes all the combined grammars as its input and extracts network signatures from the combined grammars. In the following sections, we do not introduce the first component because it is not the contribution of this papers.

3.1 Handling Network API Invocations

In this subsection, we introduce how we handle network API invocations. Our work includes two parts. The first part is building a list of network API specifications that are able to model the semantics of these APIs (modeling how they generate the network-packet contents by concatenating their arguments). We use *API grammar templates* to specify the semantics of each API, and then we can automatically generate API grammar summaries for the invocations of these APIs at runtime. An API grammar template of an API is a context-gree grammar with parameters. The parameters represent the parameters of the API, the start variable of the grammar template represents the generated network-packet content, and the productions in the grammar helps to model the semantics of the API.

To generate the list of API grammar templates, we need to manually study the possible network libraries in android system. Since we focus on HTTP network traffic in this paper, we focus on the network APIs that may generate HTTP network-packet contents. In the android library, there are mainly three sources of network APIs: Java network libraries, Apache network libraries, and Android network libraries. It should be noted that, for each android app, android system requires all its required third-party libraries to be packaged within the app. This design decision is made to help separate apps at runtime for better security of the system. It also means that it is sufficient for us to collect the network APIs in the android library. Because the byte code of all the other third-party network libraries must be packaged within the apk, and can be directly analyzed.

3.2 Combining Network APIs

Combining network APIs are typically a group of API method in a packet-generation class (e.g. Uri.builder). An instance (object) of the class, after initiated, will call methods in this group for one or more times to acquire all the data to be put into a network packet. Therefore, to handling combining network APIs, we need to trace the lifetime of packet generation instances, and build an API grammar segment for the instance as a whole. Therefore, the API grammar templates for combining network APIs are special, because it must consider the current state of the packet-generation object.

3.3 Apply String Analysis

The third component of NTApps uses string analysis to estimate the possible values of all the network arguments in the located network API invocations. String analysis is a technique to estimate the possible values of a given string variable in the code. By analyzing the data flow of string variables and string concatenations, for a given string variable v, string analysis is able to generate a context-free grammar, whose language represents the possible values of v.

3.4 Grammar Combination

The component of grammar combination, for each network API invocation *inv*, combines the API grammar summary of *inv* with the grammar of each argument of *inv*. This process is straightforward. We just replace the arguments in the API grammar summary of *inv* with the start variables of the grammar of each argument.

3.5 Extracting Network Signatures

Finally, we need to extract the network signatures from a set of combined grammars generated from the grammar combination component. The most common form of network signatures is a set of constant-string sequences, and we also leverage this form. To generate signatures of constant string sequences, we enumerate all the limited deduction trees of the grammar (i.e., we deduce only once for recursive nonterminals). Therefore, for each deduction tree, we generate a sequence of constant strings by ignoring the terminals which are not constant strings.

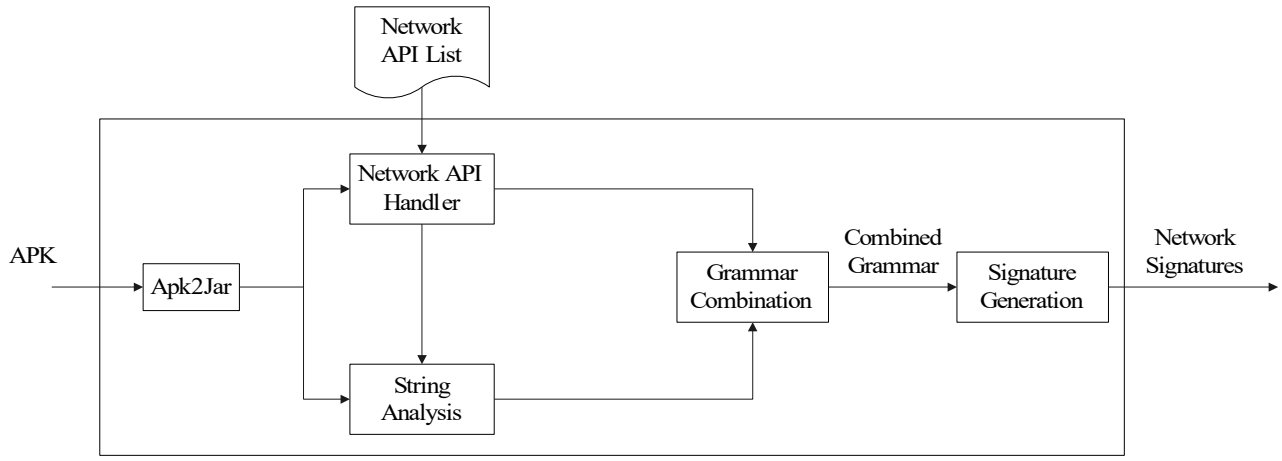

Figure 1: The overview of our approach

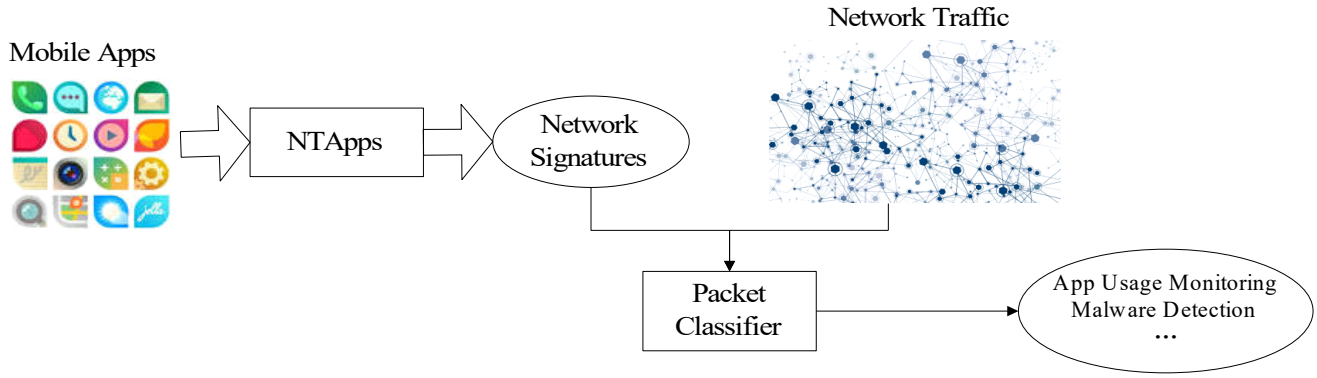

Figure 2: Usage Scenario of NTApps

Using the approach above, we can generate a set of constant-string sequences from each combined grammar. Then, we merged all these sets, and compared all these constant-string sequences to remove all the duplicate constant-string sequences. Finally, we try to extract the host name part in each constant-string sequence in the signature. It is important to extract the host name, because host names are stored separately as an item in the content of an HTTP packet, and host names are of great importance in matching network traffic. We automatically do this with the following heuristic. If the first constant string in the sequence contains both "http://" and a single '/', we extract the host name directly according to the syntax of URL. If we can find common host name postfixes (e.g., .com, .net), we extract all the constant strings after "http://" (if there is one), and before these postfixes, and then we further add these postfixes. Otherwise, we are not able to extract the host name from the constant-string sequence. If this happen, we will further check whether this constant-string sequence is trivial by checking whether there at least two constant strings in the sequence that contain letters. If not, we will remove these constant string sequences from the signature.

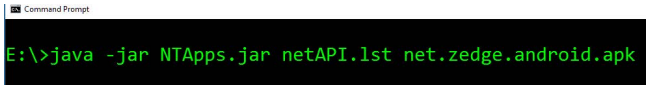

Figure 3: Input of NTApps

4 USAGE OF NTAPPS

NTApps provides a simple command line interface to allow the batch analysis of a large number of apps. As shown in Figure 3, for each app, NTApps requires just the path to the app and to the configuration file listing all the network API methods considered. Specifically, as shown in Figure 4, in the configuration file, for each API method, we specify its type (i.e., combining or simple). While a default configuration file is available with NTApps, the users have the flexibility to block certain network API methods or add more API methods in the future.

The output of NTApps is a plain text file with string constant sequences one per line, to support post-analysis filtering, signature checking, or abnormal detection. For more efficient examination in packet classifiers and users' convenience on manual inspection, we

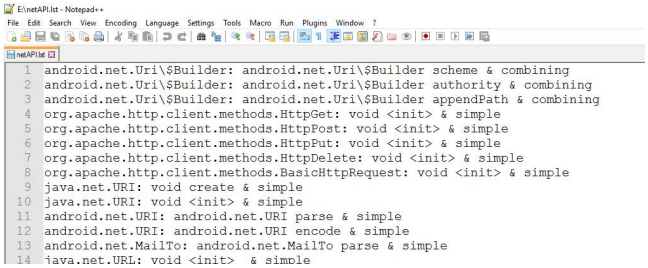

Figure 4: Configuration File with A List of Network APIs

further combine the sequences to generate a tree of common prefixes, and output the tree as dot file format for easier visualization. Part of a sample output signature tree of the app Zedge is shown in Figure 5, and part of our output signature tree for the app Soliterinc is shown in Figure 6.

5 DISCUSSION

Our approach to generate network signature is based on pure static analysis and we generate content-based signatures. Therefore, we have the following main limitations.

First of all, for encrypted network traffic such as HTTPS, we can generate the explicit signature as well. But those signatures are useless because the real network traffics are encrypted and we will never match any of them. Actually, all content-based signatures can not handle encrypted network traffic well. However, according to our statistics, more than 70% apps with Internet access do not use HTTPS. Even for the apps that use HTTPS, they often use it only in the authentication phase of the session and use HTTP for rest of the user interaction. So the HTTP signature is sufficient to identify most of the application. Our method only handle the network traffics that send out by applications.

Second, for the incoming network traffic (i.e., responses from the server), we cannot generate the corresponding signatures without the server side code. However, our evaluation shows that it is sufficient to detect apps using only outcoming network traffic (i.e., HTTP Request Packet). Furthermore, for the blocking scenario, successfully identify the outcoming network traffic is enough because once the outcoming traffic are blocked, there will be no incoming traffic.

Third, our approach is not able to handle the payloads which dynamically loaded. Actually, it is impossible to handle them statically. However, we believe that it is possible to detect the existence of dynamically loaded payload, and guide some other dynamic approach to generate such payloads.

From our evaluation, we can see that, static network signature generation, and learning-based network signature generation both have their advantages and disadvantages. The static approach is able to automatically generate network signatures for most of the apps, and it is able to cover the whole code base of the app, so it is able to find some network behaviors that are very difficult to be revealed dynamically. However, the static approach may be not precise enough in some apps, may generate invalid signatures, and cannot handle dynamically loaded payloads. In this paper, we proposed measurements to measure the quality of signatures without using them. Therefore, it is possible to first apply static approach on all the apps (since it is cheaper), and then schedule the more expensive approaches according to the quality of the statically generated signatures. The better the quality of the statically generated signature of an app is, the later can we apply more expensive approaches on it.

6 RELATED WORKS

In this section, we discuss the related works of our papers. These research efforts mainly fall into three categories: network-traffic classification, android security, and static analysis.

Network-traffic classification. A recent technique NetworkProfiler [5] is able to extract signatures with details, but it requires exhaustive exploration of the mobile apps. Statistical-information based approaches [24] [13] [2] mainly use the statistical information or the contents of the network traffic (e.g., packet size, data transferring rate, packet intervals) to perform a protocol/domain classification of network traffic. These approaches are able to identify network traffic belonging to applications of certain domains, such as database applications, video players, etc. However, similar to port-based approaches, these approaches are also coarse-grain and cannot support application-level network-traffic classification.

Content-based approaches are able to support application-level network-traffic classification by matching the payload of network packets with pre-generated signatures of specific applications. One necessary and challenging step in these approaches is to generate signatures for large number of applications. Sen et al. [34] proposed to use content-based signatures to identify the P2P network traffic of different P2P applications. These signatures are constructed manually through careful reverse engineering the P2P applications. The other group of approaches try to extract content based network signatures of an application from a large amount of network traffic of the application. There have been many efforts in this part focusing on generating the network signatures of worms from their collected network traffic. These efforts (e.g., Autograph [20], EarlyBird [35], PolyGraph [15], Hamsa [21]) basically extract common byte flows in worms' network traffic and generate a content-based signature (in the form of a string or a regular expression) for a certain worm or a group of worms. More recently, Park et al. [27] proposed to use the Longest Common Subsequence (LCS) alogrithm to generate a fingerprint of an application from the packets' content in the application's network traffic. Recently, Perdisci et al. [28] proposed a clustering-based approach to generate a signature for a group of malware sharing similar network behavior. This approach generates signatures for various HTTP-based software (not limited to worms, but also include other software applications such as adware, spyware). Although the above network-traces based signature-generation approaches are fully automatic during the signature-extraction phase. All of these efforts require a large amount of annotated representative network traffic for the application under study. Therefore, they all need manual generation of network traffic or the accumulation of network traffic from a monitored network, both of which require a relatively long time and much cost. Compared with the above approaches, our approach leverages and adapts string analysis techniques to statically generate the content-based signatures of android apps

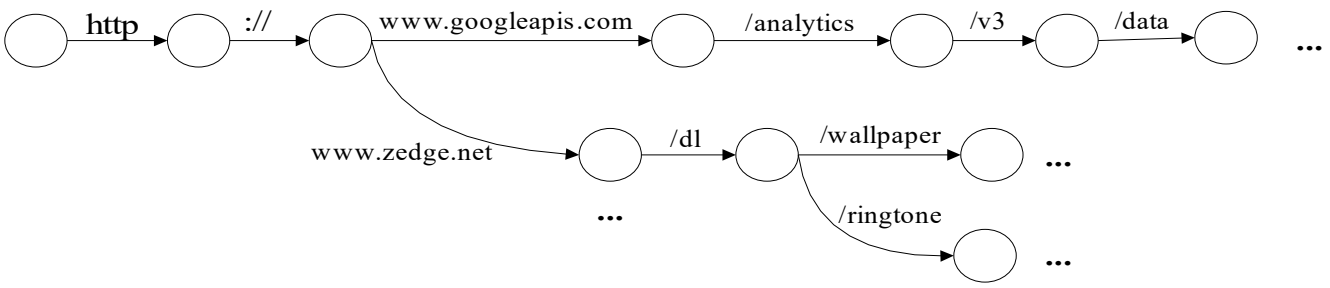

Figure 5: Output of our approach

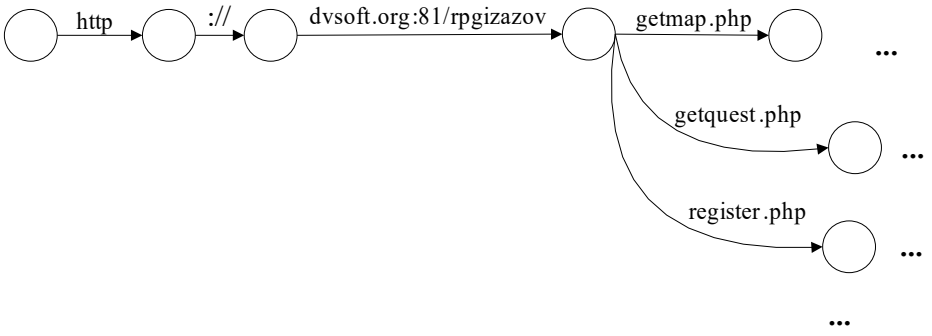

Figure 6: Another exemplar network signature

without requiring any annotated network traffic. This advantage is especially important for the signature generation of android apps because of the huge number of existing android apps and the rapid development of new android apps.

Signature generation of applications based on system level behavior (e.g., system calls) is another well studied area. Most of the approaches in this area execute the application under monitored environment and collect system event sequences as the behavior signature of the application [3] [30]. Therefore, usually, network accesses are recorded as simple system calls without considering the content sent to or received from the network. Recently, Bayer et al. [1] proposed an approach to cluster malware based on system behavior. Their approach take into account more detailed network traffic information but the considered information still only limited high-level information such as the names of downloaded files.

There are also research efforts focusing on identifying certain user-behaviors, such as browsing habits [17] or spam emails [48] [29]. These efforts usually also base their approach on a large set of network traces accumulated from real-world networks. Due to the different nature between users and apps, it is impossible and unnecessary for these efforts to provide any support for automatically generation of the network traces. So, our paper focuses on addressing a different challenge compared to these efforts.

Analysis of Mobile Applications. Our work is also related to the security analysis of mobile applications. This area is an emerging field in academic research, and some of the recent representative research efforts are presented as below. PiOS [6] is static analysis framework for iOS, which is able to check the leaking of sensitive information by combining data flow analysis and slicing techniques. Stowaway [10] is a automatic tool that is able to determine whether an Android application requests more permissions than it actually requires. The tool is based on a pre-generated mapping from Android system APIs to Android permissions. Enck et al. [9] analyzed the permission system and the permission combinations of Android System to collect a list of dangerous permission patterns and developed Kirin, a service which identifies Android application requesting dangerous permission, so that the users can be warned when installing them. Later, Enck et al. [8] further proposed ded, a de-compiler for Android application, which is able to convert Dalvik Virtual Machine code to JVM code, and then decompile the JVM code using existing Java de-compilers. As for dynamic techniques, TaintDroid [7] dynamically monitors the information flow in Android applications by tracking the propagation of taints throughout the android system. Apex [26] and TISSA [50] are two recent advancements over the current Android permission system to provide more fine-grained permission control and dynamic permission adjustability. Rocky et al. [36] proposed a novel approach to automatically detect inconsistencies between application code and privacy policies. These works mainly focus on information leaking or permissions instead of network analysis, and none of these efforts are able to generate network signatures for android apps.

Static Program Analysis. In the field of static program analysis, the research efforts that are the most related to our work is

code dependency analysis [38] [37], and especially string analysis. String analysis is a recent improvement over data-flow analysis [16]. Christensen et al. [4] first suggested string analysis, which is an approach for obtaining possible values of a string variable. Then, string analysis is widely used in various areas, especially for detecting and sanitizing SQL Injection vulnerabilities and Cross-Site-Scripting vulnerabilities. Halfond and Orso [14] used string analysis to detect and neutralize SQL injection attacks. Minamide [23] first applied string analysis on web applications. He also first suggested to simulate string operations in the extended CFG with FSTs, and implemented a string analyzer on PHP code to predict the possible values of dynamically generated web pages. Based on Minamide's work, Xie and Aiken [47] suggested a technique to detect SQL injection vulnerabilities in scripting languages. Later, Wassermann and Su first developed string-taint analysis [44] to more precisely detect the above two kinds of vulnerabilities [45]. After that, Wassermann and Su [46] further extended their work, and developed an approach to generating test cases for security vulnerabilities. Kieyzun et.al. [18] further improved their approach by considering strings that flow through the database. Wang et al. extended string analysis and developed generalized string-taint analysis [42] [41] [40], and dynamic string-taint analysis [43]. Sexena et al. [31] proposed an approach to sanitize the two kinds of vulnerabilities based on dynamic symbolic execution [32] [39] [25]. Compared to these approaches, we apply string analysis on statically generating network signatures, which is a totally different problem. We further proposed techniques to handle obfuscation and various network APIs.

7 CONCLUSIONS

In this paper, we propose a novel approach to statically generate network signatures for android apps. Our approach is based on string analysis, with adaptations to tolerate the loss of type information in the Java byte code converted from obfuscated Dalvik byte code.We further propose new techniques to handle complex network API invocations, and generating signatures from string-operation grammars. Furthermore, we introduce a formal presentation of the signature generation and matching problem, and propose the important properties of the generated network-signature set.

ACKNOWLEDGMENT

The research presented in the paper is supported in part by NSF grant CCF-1464425, and DHS grant DHS-14-ST-062-001.

REFERENCES

[1] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.
[2] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36:23–26, April 2006.
[3] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 225–238, New York, NY, USA, 2008. ACM.
[4] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS*, pages 1–18, 2003.
[5] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM*, 2013.
[6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.
[7] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.
[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
[9] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security*, pages 235–245, 2009.
[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.
[11] A. Gember, A. Anand, and A. Akella. A comparative study of handheld and non-handheld traffic in campus Wi-Fi networks. In *Passive and active measurement*, PAM, pages 173–183, Berlin, Heidelberg, 2011. Springer-Verlag.
[12] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium*, Feb. 2012.
[13] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. Acas: automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, MineNet '05, pages 197–202, New York, NY, USA, 2005. ACM.
[14] W. G. J. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. ASE*, pages 174–183, 2005.
[15] N. James, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.
[16] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, January 1976.
[17] R. Keralapura, A. Nucci, Z.-L. Zhang, and L. Gao. Profiling users in a 3g network using hourglass co-clustering. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 341–352, New York, NY, USA, 2010. ACM.
[18] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. ICSE*, pages 199–209, 2009.
[19] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *ACM CoNEXT Conference*, CoNEXT, pages 11:1–11:12, New York, NY, USA, 2008.
[20] H.-A. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.
[21] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 32–47, Washington, DC, USA, 2006. IEEE Computer Society.
[22] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected means of protocol inference. In *Conference on Internet measurement*, IMC, pages 313–326, New York, NY, USA, 2006. ACM.
[23] Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. WWW*, pages 432–441, 2005.
[24] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 50–60, New York, NY, USA, 2005. ACM.
[25] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 127–132. IEEE, 2014.
[26] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.
[27] B.-C. Park, Y. J. Won, M.-S. Kim, and J. W. Hong. Towards automated application signature generation for traffic identification. In *NOMS*, pages 160–167, 2008.
[28] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.
[29] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 291–302, New York, NY, USA, 2006. ACM.
[30] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA*, pages 108–125, 2008.
[31] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
[32] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[33] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 512–521, New York, NY, USA, 2004. ACM.

[34] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *WWW2004*, May 2004.

[35] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[36] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 25–36. ACM, 2016.

[37] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *ACM SIGPLAN Notices*, volume 50, pages 83–95. ACM, 2015.

[38] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 457–466. ACM, 2010.

[39] X. Wang, L. Zhang, and P. Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 199–210. ACM, 2015.

[40] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Transtrl: An automatic need-to-translate string locator for software internationalization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 555–558. IEEE Computer Society, 2009.

[41] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 87–96. ACM, 2010.

[42] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis. *IEEE Transactions on Software Engineering*, 39(4):516–536, 2013.

[43] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proc. FSE*, 2012.

[44] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. PLDI*, pages 32–41, 2007.

[45] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. ICSE*, pages 171–180, 2008.

[46] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proc. ISSTA*, pages 249–260, 2008.

[47] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. USENIX Security Symposium*, 2006.

[48] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 171–182, New York, NY, USA, 2008. ACM.

[49] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *SIGCOMM conference on Internet measurement conference*, IMC '11, pages 329–344, New York, NY, USA, 2011. ACM.

[50] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.