CS563: Advanced Computer Security Research Proposal

Wing Lam, Davis Li
{winglam2, dli46}@illinois.edu

October 11, 2016

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1 Introduction

What an application (or "app") *does* is not always what a user *expects*. User expectations stem from user perceptions (information they can perceive about an app). As shown in Figure 1, user perceptions come from two main sources: app descriptions form user perception before installation, and user interfaces enrich user perception after installation. Two types of inconsistencies, corresponding to each of these sources, contribute to the gap between app behavior and user perception. (1) Install-time inconsistency involves the functionalities described in app descriptions being inconsistent with the actual app behaviors. (2) Run-time inconsistency involves the behavior indicated through user interfaces (external behavior) being inconsistent with the behavior running in the background (internal behavior). In this proposal, we aim to bridge the gap between user perception and app behavior by developing a set of automated analyses to check these two inconsistencies and warn users about potential risks.

As an example of how install-time inconsistency is manifested, consider installing an app on a system with permissions that guard access to sensitive resources. The permission system from earlier versions of Android up to Android Marshmallow ask users to grant access to a list of permissions when an app is installed. Unfortunately, a list of permissions and an app description are insufficient for users to understand the potential risks in apps. A recent study [1] suggests that users find the Android permission system hard to understand because "there is no way to know what app functionality the install-time permissions correspond to." Another study [2] reveals that if users are aware of when the permission will be used in the app, they will trust the app more. For example, a majority of participants feel more comfortable using location services on navigation apps because they can perceive the use of that permission.

In more recent versions of Android (Android Marshmallow or higher), the system can ask its users to grant each permission as the permission is needed by the app. This new permission system exists so that users can decide to grant access to the requested permissions with better context as to what the app might be using them for. However, one can imagine that even with this new permission system of Android, developers with malicious intent can still deceive users to grant them access to permissions that the app may exploit. For example, an app that sends SMS messages could also upload the contacts' information to a remote malicious server. Although the context for obtaining the contact information is legitimate, the use of the contact information is unexpected. To better inform users of potential risks, we propose an undeceivable system that provides contextual information generated from *how* and *when* permissions are used by an app.

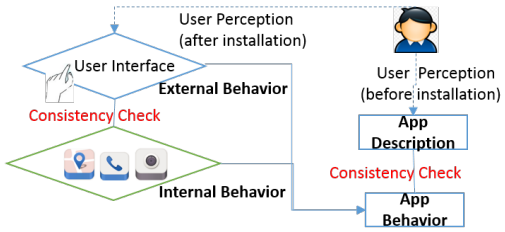

Figure 1: User perception and app behaviors

Checking install-time and run-time inconsistency faces challenges. Existing research exists for checking either inconsistency, but none of the existing work has linked the security behaviors of an app to its functional descriptions. Although some existing approaches [3, 4, 5, 6, 7, 8, 9] analyze the security attributes (e.g., the information flows from a method reading sensitive data to methods sending the data out) of an app for security analysts to review, they do not associate the app's functionalities with these security attributes. Therefore, security analysts are unable to make accurate decisions because the security attributes (e.g., information flows) in legitimate apps and illegitimate apps could be the same. On

the other hand, existing approaches [10, 11] that use data mining or natural language processing techniques to analyze an app's descriptions to identify sentences that may describe the security requirements of such apps do not consider the behavior of the app. Therefore, malware developers could easily provide fake text artifacts to fool the analysis techniques in these approaches.

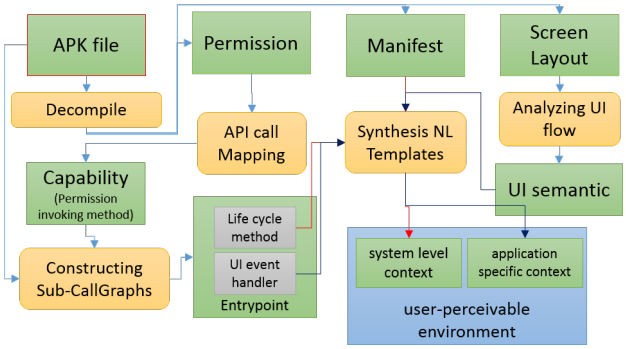

Figure 2: Overview of context recovery for permission usage

To address these challenges, we summarize our context recovery system for permission usage in the following three steps. Figure 2 depicts the overview of our context recovery system for permission usage. (1) We extract usage context by recovering the *user-perceivable environment* when each permission is used. We categorize the user-perceivable environment into two categories, the *system level* and the *application specific* context. The system level context refers to common system events such as incoming messages and phone calls, when the device starts, etc. The application specific context represents semantics of GUI actions. To recover the context information, we construct the call graph and build a mapping between the user-perceivable context and the method that directly uses a permission. Then we develop template-based natural language synthesis techniques to generate natural language descriptions of how a permission is used under what context. (2) We detect whether the underlying permission usage has additional behaviors, by performing taint analysis from each source (method that uses a permission) to identify its sinks (places where the sensitive data leave the app). For each sink method, we repeat the same procedure in our first step to synthesize natural language descriptions on permission usage context. (3) We compare the synthesized descriptions with the app descriptions using documentation-similarity analysis techniques [12, 13]. Since we aim to minimize the information overhead presented to users, we report back to users only when an inconsistency is found.

2 Background

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3 Related Work

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. [**peskin1995introduction**]

4 Proposed Approach

Our proposed approach is composed of two core components: *context recovery* and *malicious behavior checking*. The goal of the first component is to recover the context information to help users or tools to better understand the use of a permission. Our second component aims to check these malicious behaviors and report back to the user in natural language.

4.1 Recovering Context Information

The context recovery component takes as input the bytecode of the app, and outputs a list of natural language sentences explaining how a permission is used in a context. To generate the sentences, the component takes three major steps: permission refinement, context recovery, and natural language synthesis.

Permission refinement. A single Android permission often corresponds to a list of low-level capabilities. For example, by granting the permission "READ_PHONE_STATE", the app will have the capabilities to read Device Id, VoiceMail Number, Sim Serial

	Term 1	Term 2	Term 3	Term 4
Tasks	Permission usage study, Call graph construction	Context recovery, Explanation synthesis	Inconsistency checking	Evaluation, Paper writing

Table 1: Research timeline (a term represents three months)

Number, Device Software Version, etc. The coarse granularity of permissions may cause confusion when making security decisions because a single permission often represents multiple low-level capabilities. In this step, we refine the permission used in the app by detecting which capability is actually leveraged. One feasible way to get the finer-granularity capability is to utilize the method that directly uses the permission (we call it a permission-invoking method). We first employ Soot[14] to build an intra-app static call graph. Given a permission, we identify the permission-invoking method within the call graph using existing techniques [15] that build a mapping between the API and the used permission. These permission-invoking methods are typically Android library APIs, which have meaningful names and detailed descriptions in API documents. Then we can infer the used resource/capability by analyzing the API names or the API descriptions. For example, we can extract the list of low-level capabilities for a given permission (represented as one keyword per capability), and then use keyword searching against the API name or document to see which capability the API corresponds to.

Context recovery. Our definition of context includes two important characteristics that determine the invocations of security-sensitive method calls: *activation events* and *context factors*. Such definition represents a set of essential elements for decision making in app inspection.

The activation events are the *external events that trigger security-sensitive method calls*. The external events include UI events (events triggered by operations on app's user interface), System events (events initiated by system state changes (e.g., receiving SMS)), and SystemUI events (events triggered by operations on the mobile system or device interface (e.g., pressing HOME or BACK button)). Activation events connect security-sensitive behaviors to the behavior's "initiator" in the external environment (e.g., users, system), as the events are triggered when the external environment changes or reaches a certain state.

The context factors are *environmental attributes that decide whether the security-sensitive method calls will be invoked or not*. The values of context factors can affect control flows from entry points of the activation events to the security-sensitive method call. By combining the context factors with corresponding activation events of the security sensitive method call, we generate the complete context tuple.

Natural language synthesis. Finally, to present security behavior at a user-perceivable level, we synthesize natural language sentences to describe the context of the permission usage. We employ the pre-defined natural-language templates to form the skeletons of the sentences, and then automatically fill in the templates with the information obtained from the previous analysis. We have conducted a preliminary study on the permission usage of Android apps and find that there are several common patterns of permission usage. Based on these patterns, we can define templates for the permission usage context, such as making phone calls, sending emails, searching for nearby shops, as well as some UI actions. Once we generate the templates, we map the previous analysis results to each blank and complete the templates. If parts of the results are missing due to the inefficiency of the analysis, we will present user all the information that we can obtain using partial templates.

4.2 Checking Malicious Behaviors

To prevent the app from performing malicious behaviors with granted capabilities, our second component compares the actual security behaviors with the expected app behaviors to see whether the app's behavior is consistent with the expectation.

In the previous step, the synthesized natural language descriptions are not immediately sent to the user, but are first compared to the app description. We report only inconsistencies between synthesized descriptions and app descriptions as possible malicious behaviors. To achieve this goal, we first match sentences describing the same permission usage from both the synthesized description and the app description. Such matching can be done via our previous tool WHYPER [11], which leverages natural language processing techniques to locate sentences that describe a given permission in the app description. For a synthesized sentence, if we cannot locate the corresponding sentences in the app description, then it indicates that there are potential malicious behaviors in the app, and we report the additional behaviors to the user. Otherwise, if we locate the corresponding sentence in the app description, we continue to check whether the synthesized sentence is semantically consistent with the sentence in the app. To check consistency, we plan to adapt the existing documentation/sentence similarity-based techniques [12, 13] by assigning more weights to permission-related keywords based on our knowledge graph [11], and incorporate semantic information that we obtain via WHYPER, such as dependency information and part-of-speech tags.

If any inconsistency is found, we report the inconsistent behavior to the user. Otherwise, we notify the user that the app is safe.

5 Research Plan

Table 1 shows the timeline of work for the proposed research in one year. Wei Yang will focus on tasks related to Android app analysis and natural language processing; Wing Lam will take charge of program analysis and empirical study. Other work will be done jointly.

References

[1] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM. 2012, p. 3.

[2] Erika Chin et al. "Measuring user confidence in smartphone security and privacy". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM. 2012, p. 1.

[3] Adrienne Porter Felt et al. "A survey of mobile malware in the wild". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pp. 3–14.

[4] Yajin Zhou et al. "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*. 2012.

[5] Hao Peng et al. "Using probabilistic generative models for ranking risks of android apps". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 241–252.

[6] Peter Hornyack et al. "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 639–652.

[7] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *OSDI*. Vol. 10. 2010, pp. 255–270.

[8] Michael Grace et al. "Riskranker: scalable and accurate zero-day android malware detection". In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM. 2012, pp. 281–294.

[9] Yajin Zhou and Xuxian Jiang. "Dissecting android malware: Characterization and evolution". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 95–109.

[10] Mark Harman, Yue Jia, and Yuanyuan Zhang. "App store mining and analysis: MSR for app stores". In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE. 2012, pp. 108–111.

[11] Rahul Pandita et al. "WHYPER: Towards Automating Risk Assessment of Mobile Applications". In: *Proc. USENIX Security*. 2013, pp. 21–21.

[12] Yuhua Li et al. In: *Proc. FLAIRS*. AAAI Press, 2004, pp. 820–825.

[13] Xiaojun Wan, Jianwu Yang, and Jianguo Xiao. "Towards a unified approach to document similarity search using manifold-ranking of blocks". In: *Inf. Process. Manage.* 44.3 (May 2008), pp. 1032–1048. ISSN: 0306-4573. DOI: 10.1016/j.ipm.2007.07.012. URL: http://dx.doi.org/10.1016/j.ipm.2007.07.012.

[14] Christian Fritz et al. *Highly Precise Taint Analysis for Android Application*. Tech. rep. EC SPRIDE Technical Report TUD-CS-2013-0113. http://www.bodden. de/pubs/TUD-CS-2013-0113. pdf, 2013.

[15] Kathy Wain Yee Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 217–228.