A Preventive Approach for Securing Mobile Ecosystem: Assist Development of Secure Mobile Apps

Joydeep Mitra Venkatesh-Prasad Ranganath Torben Amtoft
Department of Computer Science
Kansas State University
Manhattan, KS 66506
{joydeep,rvprasad,tamtoft}@ksu.edu

ABSTRACT

Mobile apps are ubiquitous. People use them for everything from work (e.g., , email) to leisure (e.g., , games) to communication (e.g., , social media) to managing personal information (e.g., , two-step authentication). This widespread use of mobile apps in intimate ways makes them as an ideal target to cause harm to their users and the mobile ecosystem. Therefore, mobile app developers have a great responsibility to ensure their apps have no vulnerabilities. Current practices in mobile app development do not enable the developers to achieve this goal.

In this paper, we propose an app development methodology based on model driven development and storyboarding to enable developers to create apps that are void of vulnerabilities.

1. INTRODUCTION

In world of mobile apps, malicious apps exploit vulnerabilities in benign apps [4] to carry out collusion attacks, unauthorized resource usage, or private information leaks. Lack of tools to accurately (i.e., low false-positives and -negatives) identify malicious apps and the innocent-until-proven-guilty philosophy adopted by most mobile app stores (e.g., Google Play, App Store, Microsoft Store) enables malicious apps to enter app stores [8, 15]. Similar lack of tools to (help) identify vulnerabilities in mobile apps leads to developers submitting benign yet vulnerable apps for publication and app stores publishing these apps [11, 7].

In this context, *identifying malicious apps and keeping them out of the app stores* is an approach to secure the mobile ecosystem (= platform + apps). This approach has garnered immense interest in the recent past [14]. However, this approach is curative (and also reactive) as it depends on the identification of malware after it has been published and possibly affected the mobile ecosystem.

An alternative approach is to *fortify the mobile ecosystem* to thwart malicious actions. Specific to the above context, considering only the malicious apps that exploit the vulnerabilities of benign apps, an alternative approach is to *assist app developers to not create vulnerabilities in apps*. This approach is preventive (and proactive) as it preemptively eliminates vulnerabilities that could have enabled malicious apps. Also, this approach is complementary to the approach of identifying malicious apps.[1]

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. MOTIVATION

To appreciate why the alternative preventive approach is interesting, consider the following real world examples of how malicious apps exploit the vulnerabilities in benign apps to cause harm to the users of mobile devices and how malicious apps gain entry into app stores by evading malware detection systems.

2.1 Exploiting Vulnerabilities

Example 1. Camera360 is a popular application with more than 250 million downloads worldwide [5]. It is a camera app that provides users with a comprehensive set of photography options. It also provides a cloud option to store, manage, edit and share the user's photos. The app delegates a large part of its business logic to web services. While it interacts with these services using the SSL/TLS protocol, none of the application's trust managers check server certificates. As a result, a man in the middle with any certificate could impersonate the web services and set up a connection with the app. An attacker could potentially inject random images or steal users' images, steal sensitive private information like device ids and user credentials. Moreover this app also allows Javascript binding over http. As a result, an attacker could abuse private app components through the Javascript bindings and carry out privilege escalation attacks to make a phone call, send sms, email, take picture etc. The app developers were notified of these vulnerabilities and they recently released a patch without these vulnerabilities.

In this case, the vulnerabilities could have been avoided if developers had been careful when designing the app. For example, they could have used a white-list of trusted servers. They could have enforced their trust managers to check server certificates were indeed from a server in the white-list and appropriately handled cases when trust managers failed to identify a server certificate as trust-worthy. Moreover, as

[1] An alternative approach could be to detect vulnerabilities in apps submitted to app stores for publication. However, such an approach will be harder to realize as it has to deal with the source code of apps instead of possible abstractions of apps that may be easier to reason about.

they allowed Javascript binding over http, they could have used an access control-policy to manage access to components accessible via the Javascript binding.

Example 2. In 2012, it was discovered that the **Mozilla Firefox browser** on Android logged URL information visited by users [5]. Sometimes these URLs contained session IDs. Since contents written to log can be read by all apps in Android, a malicious app could have easily harvested these IDs and hijacked victim's sessions.

Recently, Android disallowed third-party apps from using the `READ_LOGS` permissions. However, sensitive information should still not be logged for apps running on older versions of Android. This is still a real threat because app developers often target their apps to run on older devices as well. Such information leaks could be avoided by considering information flow issues while designing apps.

Example 3. Another example of a vulnerability was exhibited by the system app Samsung Kies in Android [5]. This popular app allowed users to sync phone and PC. Also, this app came with all Samsung devices and could not be un-installed on un-rooted phones. It had the ability to read/write the phone's SD card and install packages on the device. It used a broadcast receiver to call an internal service that was used to read the contents of */sdcard/restore* (path in the phone's SD card), find out files with the extension *.apk* and install them on the device. If */sdcard/restore* contained files with any other extension, the service would terminate and return without changing anything. The broadcast receiver in question had an intent filter, which meant that it could be invoked from any app on the device through an intent. However, just exploiting this vulnerability would not be very malicious if the malicious app did not have write access to the SD card. To truly cause harm, a malicious app would need to inject *APK* files into the SD card. Further, Samsung Kies did not have any vulnerability either from where files could be written to SD card. However, another system app in Android called *clipboardsaveservice* had the privilege to write to SD card and also had a vulnerability. This system app contained a service that allowed any component to invoke it with the file to be written and the destination as input parameters. A malicious app could easily exploit this vulnerability to write an *APK file* of its choice. The vulnerable broadcast receiver in Samsung Kies could then be invoked to install the *APK* into the device resulting in a classic privilege escalation attack. *The root cause of the vulnerability on Samsung Kies was due to the fact that the service used to install apk files in the device was not protected enough. This could have been avoided if the developers had thought about how the service would communicate with its environment while designing the app.*

2.2 Beating Detection

While trusted repositories like Google Play continuously improve malware detection based on static and dynamic analysis, malware developers find ways to remain undetected. In this battle between application stores and malware developers, the latter are a step ahead [12]. Wang et al. [13] explain how an iOS app with a hidden buffer overflow vulnerability can escape detection and leak the user's contacts. The app requires permission to read contacts, and it is highly likely that the user will grant this permission as the app sends greeting cards to contacts on the user's phone. The app invokes a function that contacts the server to download greeting cards. This function has a hidden buffer overflow vulnerability. It then reads contacts and stores it in a variable buf. At some point in the code it overwrites buf and sends buf to the server. This is legitimate since there exists no flow where contacts flows out of the app. When the app runs, the buffer overflow vulnerability is exploited from the server to poison the stack and change the control flow of the program. In the modified control flow, the program now reads the contacts, stores it in the variable buf and jumps to the statement that sends buf to the server. Detecting such apps that change their behavior after installation is still an open challenge. Moreover, benign apps also load additional code from external sources at run-time for legitimate reasons (apk update). *If dynamic code loading from remote sources is not designed securely then malicious code can be injected to cause the benign apps to behave maliciously* [10].

3. A PROPOSAL

An alternative approach to keep malicious apps at bay is to *build an ecosystem with (almost) no vulnerabilities.* This requires preventing vulnerabilities in both mobile platforms and mobile apps. The former is a task for platform developers while the latter is a task for app developers. Almost always, there are few platforms (e.g., Android, iOS, Windows Phone) and each of them are developed and maintained by a large team with access to lots of resources. On the other hand, on each platform, there are thousands to millions of apps developed and maintained by numerous small teams with access to relatively little resources. Given this situation, the task of preventing vulnerabilities in platforms is relatively easier than the task of preventing vulnerabilities in apps as the latter requires cooperation and coordination between large number of unrelated teams. Hence, we need a solution that assists app developers to build vulnerability-free apps.

Since current app development practices do not provide assistance to prevent vulnerabilities and malicious apps that exploit vulnerabilities in other apps can enter app stores, we need to rethink about the ways to develop apps that are void of vulnerabilities.

Our proposal is to adapt and adopt an idea prevalent in software engineering and program verification communities – *enable developers to reason about non-functional requirements/properties/aspects (e.g., security) at development time and then automatically bake them into apps (as opposed to retrofitting them on to apps).*

Any realization of this proposal will need to *assist developers*

1. use and specify properties about apps,
2. reason about these properties in the context of apps,
3. ensure implementation of apps satisfy desired properties, and
4. possibly generate evidence about properties satisfied by apps.

To devise a realization of this proposal, we will briefly describe MDD and storyboarding followed by our realization of this proposal.

3.1 Model Driven Development (MDD)

In software engineering community, Model Driven Development (MDD) [6] is a well-known methodology to develop

systems by starting from an abstract model of a system and then iteratively refining the model to finally arrive at an implementation of the system. By controlling the details added in various refinement steps and using appropriate automation, developers can derive different implementations of a system starting from the same model of the system. Similarly, by considering different views (projections) of the model at various refinement steps along with appropriate automation, developers can reason about various aspects/properties of the system, e.g., security, safety, performance. By combining these abilities to reason about the model and refine the model, it is possible to derive an implementation of the system with desired properties (when feasible).

In recent years, there has been a push to develop mobile apps at a higher level of abstraction to quickly deliver the same app on different mobile platforms. In this vein, cross-platform frameworks like *Cordova* [1] and *Xamarin* [2] enable developers to develop apps in a platform independent manner and have the apps execute on different platforms with no extra development effort. In a similar vein, since MDD is well-suited hinges on the use of high-level models and supports automatic code generation for various platforms (via tooling), MDD may be well-suited for mobile app development.

The use of models and different views (projections) of models in MDD combined with automated reasoning techniques borrowed from program verification community can enable developers to reason about different aspects of apps both individually and in combination. Unlike large-scale real world programs with huge state space, mobile apps are typically componentized and each component has a relatively small contribution to the global state space. Further, instead of reasoning about the apps at source code level (which can be very hard for Java or Javascript code), with MDD, such reasoning to be performed efficiently and effectively at higher levels of abstraction.

A major hindrance to adopting MDD is that current tools and technologies that enable MDD do not easily fit into developer's work flow. In a recent empirical study of 50 practicing software developers, 35 of them stated that they do not use UML [9]. However, visual design methods like storyboards, which are gaining in popularity, can be used to enable MDD.

3.2 Storyboarding

Storyboarding is a well-known design technique in which illustrations or images are placed in order to pre-visualize scenarios. In the software world, it is used to design the work flow of software applications based on user scenarios.

Mobile app developers use storyboarding to quickly create a prototype (akin wire frame) based on which further development proceeds. Today, there are several tools to help developers create storyboards as the first step in develping mobile apps. Most notable of these tools is XCode, a IDE to develop iOS apps [3]. It allows developers to visually create all the screens of an iOS app and connect them in the order a user would navigate them. Also, it allows developers to enrich screens with data-related graphical widgets and associate them with data elements (variables) in the implementation of the app. Further, it allows the developer to execute the storyboard on a simulator or on a real device.

Clearly, given the purpose and visual inclination of storyboards, they can serve as basic models of mobile apps. With appropriate extensions to storyboards (e.g., annotations, semantics for widget actions), they can serve as models to reason about mobile apps.

3.3 The Methodology

Given the characteristics of MDD and storyboarding, we propose a methodology based on the simple combination of storyboarding and MDD – storyboarding will serve as the initial modeling technique, storyboards will serve as the initial models, and existing MDD techniques will be applied to storyboards.

4. HOW WOULD IT WORK?

In this section, we will illustrate the proposed methodology by considering a bank app as an example. The app allows users to log into the bank, select one of their accounts, and send a statement of the selected account to an email address of their choice.

Following is a "story" of how a user typically interacts with the app.

- The user launches the app by clicking on its icon on his/her mobile phone. Upon launch, the app displays the `Login` screen.
- The user enters her `Username` and `Password` and clicks on the `Submit` button. This will take the user to the `Home` screen or will keep the user in the `Login` screen. If the user's credentials are recognized, then she will be taken to the `My Accounts` screen or else she will stay on the `Login` screen.
- In the `My Accounts` Screen, the user is shown a list of accounts based on her user name. The user selects an account and clicks on the `Email Statement` button. The system will attempt to email the latest statement of the selected account to the given email address (provided by the user in the email client). If the email is sent successfully, a message is displayed to the user (on a different screen); otherwise, she stays on the `My Accounts` screen.

Figure 1 shows the initial storyboard for the app. This is similar to what app developers design initially. We will demonstrate how this model can be iteratively refined to eventually arrive at a platform specific model. The final storyboard will look like the one shown in Figure 2. But we will not achieve that before a few iterations.

Iteration 1 (Control Flow). The storyboard in Figure 1 captures all possible transitions an app can make. It does not say what happens when the user performs an action, e.g., clicks a button. For example, the storyboard only indicates that two transitions are possible from the `Login` screen. It does not indicate when those transitions will be triggered. The story of the app outlined above suggests that both these transitions are possible when the user clicks on the `Submit` button. In Figure 2 user actions are depicted as *transition labels* of the form α_{action}(Label). At this point, Figure 2 will look like Figure 1 only with the transitions labeled with user action specifications, e.g., $\alpha_{launch}()$ and $\alpha_{click}(Signin)$. Also, we can verify the design for correctness, e.g., `MyAccounts` will be displayed if and only if the user was on `Login` screen and had clicked the `SignIn` button.

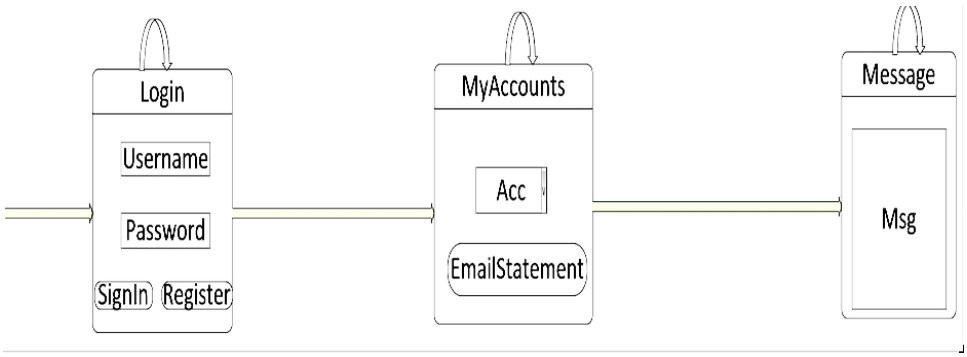

Figure 1: Initial Storyboard.

Iteration 2 (Guarded Control Flow). In the user story outlined above, some transitions depend on user actions and some additional constraints. We refer to such additional constraints that are independent of user actions as *predicates* (denoted by ϕ). A transition is enabled only if the associated predicate evaluates to true. In Figure 2, a transition from `Login` screen to `My Accounts` screen is enabled if the action has been performed in the `Login` screen and if the credentials have been recognized. The predicate $\phi_{=}(\lambda_{private}$**(LOGIN_SERV, get,Login.Username.info)**, *Login.Password.info*) is used to represent the action required to recognize the credentials. The predicate $\phi_{=}$ takes two arguments and checks for their equality. The first argument $\lambda_{private}$*(LOGIN_SERV, get, Login.Username.info)* denotes the retrieval of the password for the provided user name from an external component identified by `LOGIN_SERV`. The second argument `Login.Password.info` denotes the password entered by the user. The `private` keyword indicates the λ expression can be invoked only from within the app.

At this point, the refined model can be verified by evaluating it for previously checked properties and new properties. For example, the previously checked property that `MyAccounts` can only appear if the user was on `Login` and had clicked on the `Sign In` button should hold true even now. We can additionally assert that `MyAccount` can appear only if the user's username and password have been verified.

Iteration 3 (Information Flow). In `My Accounts` screen, the user should be able to see a list of accounts based on her user name. For this purpose, the user name entered in `Login` screen should be available in this screen. This is specified by associating the data flow requirement `{u=Login.Username.Info}` (in orange in Figure 2) with the transition from `Login` screen to `My Accounts` screen.

Such information flow specification can help check if the app leaks sensitive information. For example, the developer might want to assert the app never leaks the user name. At this point, based on the information in the storyboard, the methodology will deduce the app is leaking the user name into `LOGIN_SERV` (as it is unclear if the user name can be sent to `LOGIN_SERV`).

Iteration 4 (More Information Flow). At this point, there is no information about (1) what the user will see on the `My Accounts` screen or (2) what can she do when she is on `My Accounts` screen.

With regards to (1), according to the textual story, in `My Accounts` screen, she should see a list of accounts that she owns. The specification *Acc.Info* = $\lambda_{private}$*(ACC_SERV,get,u)* (in Figure 2) provides the information required by (1). The specification means the information in `Acc` widget is initialized with the information retrieved from `ACC_SERV` based on the user name (captured by `Login.Username.info`).

With regards to (2), the user will transition from `My Accounts` screen to `Message` screen if (a) she clicks on the `Email Statement` widget, (b) the latest statement for the selected account could be retrieved and saved to a file on the device, and (c) the saved file was successfully emailed to the provided address. Constraint (b) is denoted by $\lambda_{private}$*(FILE_PATH, save,*$\lambda_{private}$*(STMT_SERV,get,MyAccounts.Acc.selectedInfo))* (in Figure 2) where `STMT_SERV` denotes the component from where the statement is retrieved and `FILE_PATH` is the path where the statement is saved. The expression evaluates to true only if the statement was saved successuflly. Constraint (c) is denoted by $\lambda_{private}$*(EMAIL,invoke,FILE_PATH)* where `EMAIL` is the email client used to email the statement saved at `FILE_PATH`. This expression evaluates to true only if the email is sent successfully.

At this point we verify some additional properties. Note that the earlier properties should hold as well. For example, we can assert security properties like an email can be sent by this app only if the user explicitly requests for it. A manual analysis reveals that this property indeed holds true because the only way to invoke EMAIL is if the transition from MyAccounts to Message is enabled. For that to happen the user must click on *EmailStatement*.

Iteration 5 (Information Classification). So far, all information was assumed to be sensitive and every external component was assumed to be untrusted. This prohibited accurate information flow analysis. For example, in Iteration 3 the analysis deduced that user name was getting leaked to LOGIN_SERV, but LOGIN_SERV is a trusted component so it should be allowed. To avoid this situation, in this iteration, only sensitive information is tagged as *high(h)* while all other information is (implicitly) tagged as *low(l)*. Similarly, external components are tagged as *trusted(t)* and *untrusted(u)* as shown in Figure 2. With these annotations, various information flow properties can now be verified accurately. For example, we can confirm the app does not leak user name to untrusted regions.We can further assert that sensitive information never flows into untrusted regions.

However, in this model, there exists a information flow

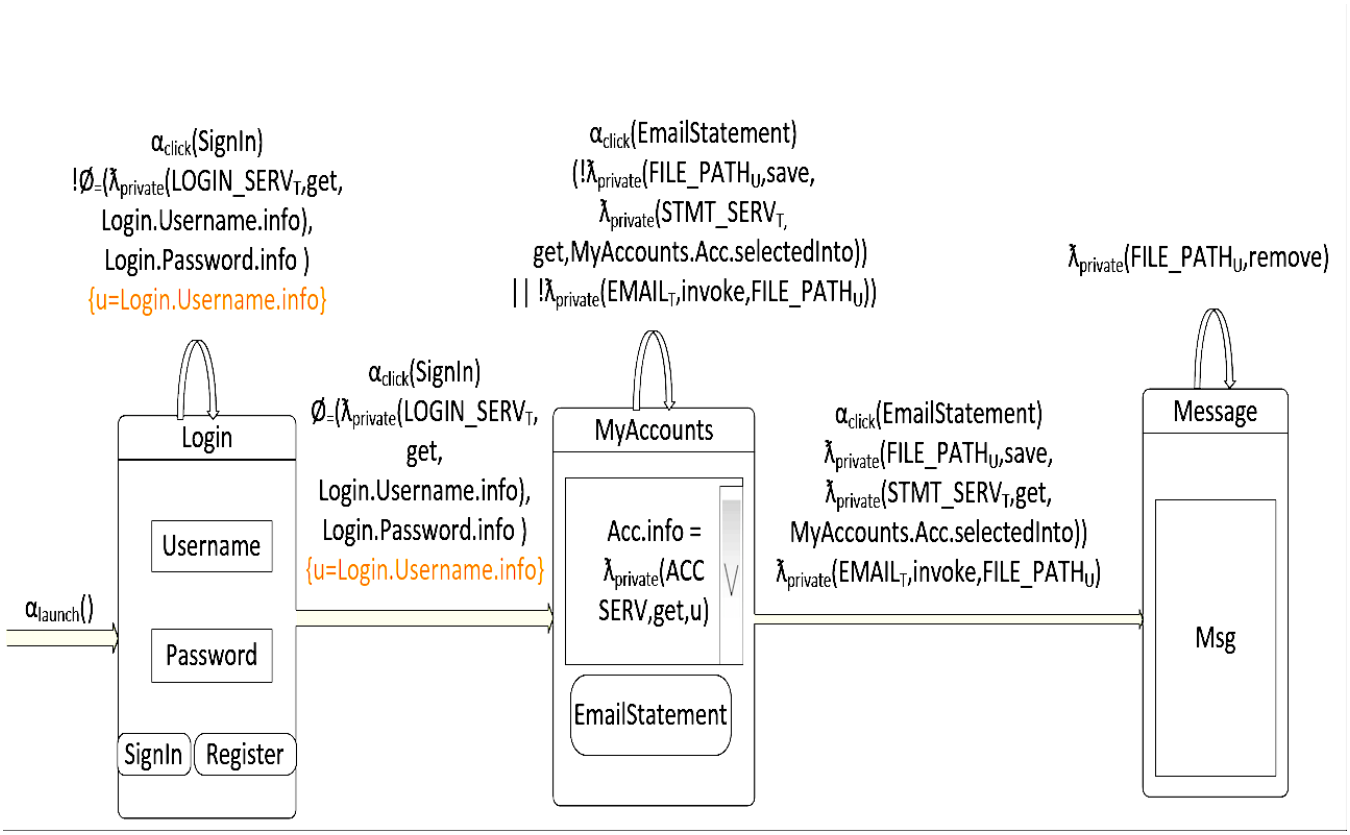

Figure 2: Final Storyboard.

path along which the sensitive information *account statement* flows into an untrusted sink `FILE_PATH`. Assuming there is no more information available about account statement and `FILE_PATH` that can influence the information flow, a design flaw that leads to a security vulnerability has be uncovered and needs to be addressed, e.g., by encrypting the account statement or changing `FILE_PATH` to be in a trusted region of storage accessible only to the app and the email client. This trust chain can be extended further to only involve trusted email clients (e.g., white-listed) by annotating the corresponding lambda expression to invoked trusted email clients (denoted by the subscript T on `EMAIL`).

Finally, we will arrive at a storyboard as shown in Figure 2. From this model, implementations (source code) that embody various checks and computations (as indicated by the annocations) required to exhibit the desired properties can be generated by the push of a button.

5. CHALLENGES

While the proposed approach is interesting and can be immensely useful, the following challenges (tasks) need to be addressed (tackled) to realize the approach and foster its adoption in the community.

1. Design an accessible visual language to describe storyboards and annotate them with properties that can be used for automated reasoning.

2. Extend existing tools and technologies or develop new ones to support the proposed approach, e.g., model analysis, property verification, code generation.

3. Evaluate if and to what extent does the approach help develop secure mobile apps and eventually secure mobile ecosystems. In addition, identify the class of properties that are (not) addressed best by the approach.

4. Evaluate if and to what extent does the approach affect the productivity of app developers. Depending on when and how this evaluation is done, it could inform the realization of the approach.

5. Evaluate if and to what extent the approach delivers on its promise of traceability – properties satisfied by the model will be also be satisfied by the final implementation derived from the model.

6. Collect and share the set of properties that is common across multiple apps on a mobile platform.

7. Evaluate how effective are the set of communal properties in securing the apps and the ecosystem.

8. Generate machine checkable evidence for the properties satisfied by the apps that can be used to by app stores to review apps before publication.

6. REFERENCES

[1] Documentation for building apps with cordova. Available at https://cordova.apache.org/docs/en/latest/.

[2] Documentation for building apps with xamarin. Available at https://developer.xamarin.com/guides/.

[3] Apple. Cocoa application competencies for ios - storyboard. Available at Apple Developer Documentation.

[4] E. Chin. Helping developers construct secure mobile applications, 2013. Available at https://escholarship.org/uc/item/4x48p6rz#page-1.

[5] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. Wiley Publishing, 1st edition, 2014.

[6] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54. IEEE Computer Society, 2007.

[7] M. Gomez, R. Rouvoy, M. Monperrus, and L. Seinturier. A Recommender System of Buggy App Checkers for App Store Moderators, 2014. Available at Online Tech Report.

[8] Y. Kikuchi, H. Mori, H. Nakano, K. Yoshioka, T. Matsumoto, and M. van Eeten. Evaluating malware mitigation by android market operators. In *9th Workshop on Cyber Security Experimentation and Test (CSET 16)*. USENIX Association, 2016.

[9] M. Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 722–731. IEEE Press, 2013.

[10] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. Citeseer, 2014.

[11] A. Sadeghi, N. Esfahani, and S. Malek. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *International Conference on Fundamental Approaches to Software Engineering*, pages 155–169. Springer, 2014.

[12] V. V. T. Tong, J.-F. Lalande, and M. Leslous. Challenges in android malware analysis. Number 106, pages 42–43, 2016.

[13] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 559–572. USENIX, 2013.

[14] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Comput. Surv.*, 49(2):38:1–38:47, 2016.

[15] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.