Challenges for Dynamic Analysis of iOS Applications

Martin Szydlowski[1], Manuel Egele[2], Christopher Kruegel[2], and Giovanni Vigna[2]

[1] Secure Systems Lab, Vienna University of Technology, Austria
msz@seclab.tuwien.ac.at
[2] University of California, Santa Barbara
{maeg,chris,vigna}@cs.ucsb.edu

Abstract. Recent research indicates that mobile platforms, such as Android and Apple's iOS increasingly face the threat of malware. These threats range from spyware that steals privacy sensitive information, such as location data or address book contents to malware that tries to collect ransom from users by locking the device and therefore rendering the device useless. Therefore, powerful analysis techniques and tools are necessary to quickly provide an analyst with the necessary information about an application to assess whether this application contains potentially malicious functionality.

In this work, we focus on the challenges and open problems that have to be overcome to create dynamic analysis solutions for iOS applications. Additionally, we present two proof-of-concept implementations tackling two of these challenges. First, we present a basic dynamic analysis approach for iOS applications demonstrating the feasibility of dynamic analysis on iOS. Second, addressing the challenge that iOS applications are almost always user interface driven, we also present an approach to automatically exercise an application's user interface. The necessity of exercising application user interfaces is demonstrated by the difference in code coverage that we achieve with (69%) and without (16%) such techniques. Therefore, this work is a first step towards comprehensive dynamic analysis for iOS applications.

1 Introduction

Mobile devices and especially smart phones have become ubiquitous in recent years. They evolved from simple organizers and phones to full featured entertainment devices, capable of browsing the web, storing the user's address book, and provide turn-by-turn navigation through built in GPS receivers. Furthermore, most mobile platforms offer the possibility to extend the functionality of the supported devices by means of third party applications. Google's Android system, for example, has the official Android Market [1], and a series of unofficial descendants that provide third party applications to users. Similarly, Apple initially created the AppStore for third party applications for their iPhones. Nowadays, all devices running iOS (i.e., iPhone, iPod Touch, and iPad) can access and

J. Camenisch and D. Kesdogan (Eds.): iNetSec 2011, LNCS 7039, pp. 65–77, 2012.

download applications from the AppStore. To ensure the quality and weed out potentially malicious applications, Apple scrutinizes each submitted application before it is distributed through the AppStore. This vetting process is designed to ascertain that only applications conforming to the iPhone Developer Program License Agreement [3] are available on the AppStore. However, anecdotal evidence has shown that this vetting process is not always effective. More precisely, multiple incidents have become public where applications distributed through the AppStore blatantly violate the user's privacy [6, 16], or provide functionality that was prohibited by the license agreement [26]. After Apple removed the offending applications from the AppStore no new victims could download these apps. However, users that installed and used these apps prior to Apple noticing the offending behavior had to assume that their privacy had been breached.

Recent research [9, 13] indicates that AppStore applications regularly access and transmit privacy sensitive information to the Internet. Therefore, it is obvious that the current vetting process as employed by Apple requires improvement. With static analysis tools available to investigate the functionality of malicious applications, one has to assume that attackers become more aware of the risk of getting their malicious applications identified and rejected from the AppStore. Thus, we assume attackers become more sophisticated in hiding malicious functionality in their applications. Therefore, we think it is necessary to complement existing static analysis techniques for iOS applications with their dynamic counterparts to keep the platform's users protected. We are convinced that the combination of static and dynamic analysis techniques make a strong ensemble capable of identifying malicious applications. To this end, this paper makes the following contributions:

- We highlight the challenges that are imposed on dynamic analysis techniques when targeting a mobile platform such as iOS.
- We implement and evaluate a dynamic analysis approach that are suitable for the iOS platform.
- We create an automated system that exercises different aspects of the application under analysis by interacting with the application's user interface.

2 Dynamic Analysis

Dynamic analysis refers to a set of techniques that monitor the behavior of a program while it is executed. These techniques can monitor different aspects of program execution. For example, systems have been developed to record different classes of function calls, such as API calls to the Windows API [25], or system calls for Windows [8] or Linux [20]. Systems performing function call monitoring can be implemented at different layers of abstraction within the operating system. For example, the JavaScript interpreter of a browser can be instrumented to record function and method calls within JavaScript code [17]. Dynamic binary rewriting [18] can be leveraged to monitor the invocation of functions implemented by an application or dynamically linked libraries. Similarly, debugging

mechanisms can be employed to gather such information [22, 23, 24]. Furthermore, the operating system used to perform the analysis might provide a useful hooking infrastructure. Windows, for example, provides such hooks for keyboard and mouse events. The dtrace [2] infrastructure available on Solaris, FreeBSD, and Mac OS X can also be used to monitor system calls.

An orthogonal approach to function call monitoring is information flow analysis. That is, instead of focusing on the sequence of function calls during program execution, the focus is on monitoring how the program operates on interesting input data [7]. This data could, for example, be the packets that are received from the network, or privacy relevant information that is stored on the device. By tracking how this data is propagated through the system, information flow monitoring tools can raise an alert if such sensitive data is about to be transmitted to the network [10, 27]. In the case of incoming network packets the same technique can be applied to detect attacks that divert the control flow of an application in order to exploit a security vulnerability [21].

3 Challenges for Dynamic Analysis on the iOS Platform

State of the art. Existing dynamic analysis techniques are geared towards applications and systems that execute on commodity PCs and operating systems. Therefore, a plethora of such systems are available to analyze x86 binaries executing on Linux or Windows. While the x86 architecture is most widely deployed for desktop and server computers, the landscape for the mobile device market has a different shape. In the mobile segment, the ARM architecture is most prevalent. The rise of malicious applications [15] for mobile platforms demands for powerful analysis techniques to be developed for these systems to fight such threats. However, existing dynamic analysis techniques available for the x86 architecture are not immediately applicable to mobile devices executing binaries compiled for the ARM architecture. For example, many dynamic analysis approaches rely on full system emulation or vitalization to perform their task. For most mobile platforms, however, no such full system emulators are available. While Apple, for example, includes an emulator with their XCode development environment, this emulator executes x86 instructions, and therefore requires that the application to emulate is recompiled. Thus, only applications that are available in source code can be executed in this emulator. However, the AppStore only distributes binary applications, which cannot be executed in the emulator. Furthermore, the emulator's source code is not publicly available, and therefore, cannot be extended to perform additional analysis tasks. OS X contains the comprehensive `dtrace`[1] instrumentation infrastructure. Although the iOS kernels and OS X kernels are quite similar iOS does not provide this functionality.

Graphical user interfaces (GUI). An additional challenge results from the very nature of iOS applications. That is, most iOS applications are making heavy use of event driven graphical user interfaces. Therefore, launching an application and

[1] `http://developers.sun.com/solaris/docs/o-s-dtrace-htg.pdf`

executing the sample for a given amount of time might not be sufficient to collect enough information to assess whether the analyzed application poses a threat to the user or not. That is, without GUI interaction only a minimal amount of execution paths will be covered during analysis. Therefore, to cover a wide range of execution paths, any dynamic analysis system targeting iOS applications has to be able to automatically operate an applications' GUI.

Source vs. binary analysis. Combined static and dynamic analysis approaches, such as Avgerinos et al. [4], can derive a semantically rich representation of an application by analyzing its source code. However, applications distributed through the AppStore are available in binary form only. Therefore, any analysis system that targets iOS applications can only operate on compiled binaries.

Analyzing Objective-C. The most prevalent programming language to create iOS applications is Objective-C. Although Objective-C is a strict superset of the C programming language it features a powerful runtime environment that provides functionality for the object-oriented capabilities of the language. With regard to analyzing a binary created from Objective-C it is especially noteworthy that member functions (i.e., methods) of objects are not called directly. Instead, the runtime provides a dynamic dispatch mechanism that accepts a pointer to an object and the name of a method to call. The dispatch function is responsible for traversing the object's class hierarchy and identifying the implementation of the corresponding method.

The above mentioned problems combined with the constraint hardware resources of mobile devices pose significant challenges that need to be addressed before a dynamic analysis system for the iOS platform becomes viable.

4 Strategies to Overcome these Challenges

To tackle the above mentioned challenges, this work makes two major contributions. First, we implement and evaluate a dynamic analysis approach that is suitable for the iOS platform and provides a trace of method calls as observed during program execution. Second, we create an automated system that exercises different functionality of the application under analysis by interacting with the application's user interface.

4.1 Dynamic Analysis Approaches

As mentioned above not all dynamic analysis techniques available on the x86 architecture are feasible on iOS devices executing ARM instructions. Although there are many different approaches to dynamic analysis, we think that function call traces are a viable first step in providing detailed insights into an application's behavior. Therefore, in this section we elaborate on the lessons we learned while implementing a system that allows us to monitor the invocation of function calls of iOS applications.

Objective-C is the most prevalent programming language used to create applications for the iOS platform. However, as opposed to C++ where methods (i.e., class member functions) are invoked via the use of *vtable* pointers, in Objective-C methods are invoked fundamentally different. More precisely, methods are not called but instead a so-called *message* is sent to a receiver object. These messages are handled by the dynamic dispatch routine called `objc_msgSend`. This dispatch routine is responsible for identifying and invoking the implementation for the method that corresponds to a message. The first argument to this dispatch routine is always a pointer to the so-called *receiver* object. That is, the object on which the method should get invoked (e.g., an instance of the class `NSMutableString`). The second argument is a so-called *selector*. This selector is a string representation of the name of the method that should get invoked (e.g., `appendString`). All remaining arguments are of no immediate concern to the dispatch function. That is, these arguments get passed to the target method once it is resolved. To perform this resolution, the `obj_msgSend` function traverses the class hierarchy starting at the receiver and searches for a method whose name corresponds to the selector. Should no match be found in the receiver class, its superclasses are searched recursively. Once the corresponding method is identified, the dispatch routine invokes this method and passes along the necessary arguments. Due to the prevalence of Objective-C to create iOS applications we chose to implement a dynamic analysis approach that monitors the invocation of Objective-C methods instead of classic C functions.

Monitoring the dispatcher. One approach of monitoring all method invocations through the dynamic dispatch routine would be to hook the dispatch function itself. This could be achieved by following an approach similar to Detours [18]. That is, one would copy the initial instructions of the dynamic dispatcher before replacing them in memory with an unconditional jump to divert the control flow to a dedicated hook function. This hook function could then perform the necessary analysis, such as resolving parameter values, and logging the method invocation to the analysis report. Once the hook function finished executing, control would be transferred back to the dispatch function and regular execution could continue. Of course, the backed up initial instructions that got overwritten in the dispatcher need to be executed too before control is transferred back to the dispatch function. Although such an approach seems straight forward, the comprehensive libraries available to iOS applications also make extensive use of the Objective-C runtime. Therefore, such a generic approach would collect function call traces not only on the code the application developer created but also on all code that is executed within dynamically linked libraries. Often, however, function call traces collected from libraries are repetitive. Thus, we chose to implement our approach to only trace method invocations that are performed by the code the developer wrote.

Identifying method call sites. As a first step in monitoring Objective-C method calls we leverage our previous work PiOS [9] to generate a list of call sites to the dynamic dispatch function. Furthermore, PiOS is often capable of determin-

ing the number and types of arguments that are passed to the invoked method. This information is recorded along with the above mentioned call sites. Subsequently, this information is post processed to generate `gdb`[2] script files that log the corresponding information to the analysis report. More precisely, for each call site to the dynamic dispatch function, the script will contain a breakpoint. Furthermore, for each breakpoint hit the type (i.e., class) of the receiver as well as the name of the invoked method (i.e., the selector) get logged. Additionally, if PiOS successfully determined the number of arguments and their types, this information will also be logged.

4.2 Automated GUI Interaction

Most iOS applications feature a rich graphical user interface. Furthermore, most functionality within those applications gets executed in response to user interface events or interactions. This means that unless an application's user interface is exercised, most of the functionality contained in such applications lies dormant. As dynamic analysis only observes the behavior of code that is executing, large parts of functionality in such applications would be missed unless the GUI gets exercised.

Therefore, one of the challenges we address in this work is the automated interaction with graphical user interfaces. Such interaction with an application's GUI can be achieved on different levels. Desktop operating systems commonly support tools to get identifiers or handles for currently displayed GUI elements (e.g., UI explorer on Mac OS X). However, no such system is readily available for iOS.

Therefore, we turned our attention to alternative solutions to exercise an application's user interface. A straight forward approach could, for example, randomly click on the screen area. This method proved effective in detecting click-jacking attacks [5] on the World Wide Web. A more elaborate technique could read the contents from the device's frame buffer and try to identify interactive elements, such as buttons, check-boxes, or text fields by applying image processing techniques. Once such elements are identified, virtual keystrokes or mouse clicks could be triggered in the system to interact with these elements. We combined these two approaches into a proof-of-concept prototype that allows us to automatically exercise graphical user interfaces of iOS applications.

To interact with the device and get access to the device's frame buffer we leverage the open source Veency[3] VNC server. To communicate with the VNC server, and perform the detection and manipulation of UI elements, we have modified the python-vnc-viewer[4], an open source VNC client implementation in Python.

The basic idea behind this approach is to sample the screen and *tap* (i.e., click) locations on the screen that are determined by a regular grid pattern.

[2] `http://www.gnu.org/s/gdb/`
[3] `http://cydia.saurik.com/info/veency/`
[4] `http://code.google.com/p/python-vnc-viewer/`

Additionally, to identify interactive user interface elements, we perform the following steps in a loop: We capture the contents of the screen buffer and compare it to the previous screenshot (if present). If a sufficiently large fraction of pixels has changed between the images, we assume an interactive element has been hit. To tell input fields from other interactive elements apart, the current screenshot is compared to a reference image where the on-screen keyboard is displayed. This comparison is based on a heuristic that allows slight variations in the keyboard's appearance (e.g., different language settings). If we can determine that a keyboard is displayed, we send tap events to the first four keys in the middle row (i.e., `ASDF` on a US layout) and the return/done key to dismiss the keyboard again. When no keyboard is detected, we advance the cursor to the next location and send a tap event. In either case, we wait a brief amount of time before repeating the procedure, to give the UI time to respond and complete animations. We empirically determined a wait time of 3 seconds to be sufficient.

To avoid hitting the same UI elements repeatedly, we keep a greyscale image with dimensions identical to the frame buffer in memory. We call this greyscale image a *clickmap*. For each tap event, we perform a fuzzy flood-fill algorithm on the screenshot, originating from the tap coordinates, to determine the extents of the element we have tapped. That approach works well for monochrome or slightly shaded elements, like the default widgets offered by the interface builder for iOS applications. We mark these extents in the clickmap to keep track of the elements we have already accessed. That is, before a tap event is actually sent, the clickmap is consulted. If the current coordinates belong an element we already clicked, no tap event will be sent. Therefore, we avoid hitting the same element repeatedly, especially when the element in question is the background. Whenever we have a new screenshot containing changes, we clear the changed area in the clickmap so that new UI elements that might have appeared will be exercised too.

5 Evaluation

In this section we present the results we obtained during the evaluation of our prototype implementation. For the purpose of this evaluation we created a sample application that contains different user interface components such as buttons, text fields, and on/off switches. A screenshot of the application is depicted in Figure 1. The rationale for creating such a sample application is that by creating the application, we got intimately familiar with its functionality and operation. Furthermore, our experience with the static analysis of iOS applications allowed us to build corner cases into the application where we know static analysis can only provide limited results. For example, the test application would dynamically generate a new text field, once a specific button is clicked.

5.1 Method Call Coverage

PiOS identified a total of 52 calls to the dynamic dispatch function. Once our test application is launched, no further method calls are made unless the different user

Fig. 1. A screenshot of the sample application. The lower text field is dynamically created upon the first click to the *Reset* button.

interface elements are exercised. This is common behavior for iOS applications that are heavily user interface driven. During application startup only 8 of the 52 method calls (i.e., 16%) are executed. This underlines that dynamic analysis approaches that do not take the GUI of an iOS application into account, can only provide limited information about the application's functionality. Moreover, the methods that can be observed during program startup are generically added by Apple's build system and are almost identical for all applications targeting the iOS platform. Therefore, the valuable insights into application behavior that can be derived solely from the program startup phase are limited at best.

By executing our prototype to exercise the user interface of our test application, we could observe 36 methods being called. That corresponds to 69% of all methods being covered. Most importantly, we were able to exercise most of the functionality that is not part of the initial startup procedures for the applications. Our system did not observe the remaining 16 methods being invoked. One method would only be called if the user interface was in a very specific state.[5] However, our technique to exercise the user interface did not put the application in that state. All remaining 15 calls were part of the shutdown procedures (e.g., destructors) for the application. However, these methods are only invoked if the application terminates voluntarily. If the user presses the home button on the device, the application is terminated and no cleanup code is executed. As there

[5] If the switch has been switched from the default *on* setting to *off* and the *Reset* button is clicked afterwards, a message is sent to animate the switch back to its default *on* setting.

is not generic way to determine whether a certain user interface element will exit an application, our analysis terminates the application by tapping the home button. Thus, we did not observe these shutdown procedures being executed.

5.2 Comparison with Static Analysis

There are different possibilities on how to compare the static and dynamic analysis results. For example, static analysis covers all possible and thus also infeasible execution paths. Dynamic analysis can only observe the code paths that are executed while the program is analyzed. Therefore, we first evaluate how many and which methods get invoked during dynamic analysis. To compare the dynamic and static analysis results we first analyzed our test application with PiOS.

Static analysis results. PiOS detected 52 calls to the `objc_msgSend` dynamic dispatch function. In 49 cases (i.e., 94%) PiOS was able to statically determine the class of the receiver object and the value of the selector. Furthermore, PiOS validates these results by looking up the class of the receiver in the class hierarchy. A method call is successfully resolved if the class exists in the class hierarchy and this class or one of its superclasses implements a method whose name corresponds to the value of the selector.

The remaining three method calls that PiOS was unable to resolve are part of the function that dynamically creates and initializes the new text field. In our sample application this action is performed the first time the *Reset* button is clicked.

Comparison. We compared the receiver and selector for the 36 method calls present in the static and dynamic analysis reports. In all but 3 instances the results were identical. In two of these three instances PiOS identified the receiver type as `NSString`, whereas the dynamic analysis indicates that the actual type is `CFConstantStringClassReference`. However, according to Apple's documentation[6] these two types can be used interchangeably. In the third instance PiOS identified the receiver as `NSString` and dynamic analysis indicates the correct type to be `NSPlaceHolderString`. The difference is that for `NSPlaceHolderString` the initialization is not complete yet. This inconsistency is plausible as the only time this happened is in a call to `initStringWithFormat` to finish initialization.

5.3 Method Call Arguments

For 12 calls PiOS was able to determine the types of the arguments that get passed to the invoked methods. Thus, in these cases the dynamic analysis script is also logging information pertaining to these arguments. More precisely, for arguments of type `NSString` or any of its related types, a string representation of the argument is printed in the log file. For all other types, the address of the corresponding argument is printed instead.

[6] `http://developer.apple.com/library/mac/#documentation/CoreFoundation/Reference/CFStringRef/Reference/reference.html`

5.4 Improvements for Static Analysis

As mentioned previously, static analysis is sometimes unable to compute the target method of an `objc_msgSend` call. More precisely, if PiOS is unable to statically determine the type of the receiver object or the value of the selector, PiOS cannot resolve the target method. This was the case for 3 method calls in our sample application. All 3 instances were part of the function that dynamically creates the additional text field. However, the dynamic analysis exercised this functionality and the analysis report contains the type of the receiver object and the value of the selector. Thus, the results from dynamic analysis can be leveraged to increase precision of our static analysis system.

6 Limitations and Future Work

Our proof-of-concept implementation relies on `gdb` to collect information about the program during execution. Thus, it is easily detectable by applications that try to detect whether they are analyzed. Therefore, we plan to evaluate stealthier ways of performing the necessary monitoring tasks in the future. Furthermore, our current implementation does not take system calls into account. However, for example Cydia's *mobile substrate* framework could be a good starting point to investigate system call monitoring on iOS devices.

Although our method of exercising the user interface resulted in high code coverage when compared to the functionality of program startup alone, we see room for improvement in this area too. That is, our current approach does not handle highly non-uniform colored (i.e., custom designed) user interface elements correctly. More precisely, our system does not detect the boundaries of such elements reliably and therefore might tap the same element multiple times. Furthermore, we only consider tap events and omit all interactions that use swipe or multi touch gestures. Therefore, to improve the automatic user interface interaction, one could try to extract the information about UI elements from the application's memory during runtime. This would entail getting a reference to the current `UIView` element and find a way to enumerate all UI elements contained in that view. We plan to investigate such techniques in future work.

7 Related Work

The wide range of related work in dynamic analysis mainly focuses on desktop operating systems. Due to the challenges mentioned above these techniques are not readily applicable to mobile platforms. For brevity, we refer the reader for such techniques to [11] and focus this section on related work that performs analysis for mobile platforms. TaintDroid [12] is the first dynamic analysis system for Android applications. However, it is limited to applications that execute in the Dalvik virtual machine. Thus, by modifying the open source code of the virtual machine, the necessary analysis steps can be readily implemented. However, the authors state that "Native code is unmonitored in TaintDroid".

Therefore, systems like TaintDroid are not applicable to the iOS platform as iOS applications execute on the hardware directly. That is there is no middle-layer that can be instrumented to perform analysis tasks.

Mulliner et al. [19] use labeling of processes to prevent cross-service attacks on mobile devices. However, this approach relies on a modified Linux kernel to check and verify which application is accessing which class of devices. Such checks only reveal the communication interfaces that are used by an application. Applications on the AppStore, however, are prevented from accessing the GSM modem, and thus can only access the network or Bluetooth components. Thus, such a system is too coarse grained to effectively protect iOS users. Furthermore, the source code for iOS is not available, and thus the necessary modifications to the operating systems' kernel cannot be made easily.

In previous work we presented PiOS [9] as an approach to detect privacy leaks in iOS applications using static analysis. This work demonstrated that it is indeed common for applications available in the AppStore to transmit privacy sensitive data to the network – usually without the users consent or knowledge. Furthermore, Enck et al. [14] presented Kirin a system that statically assesses, whether the permissions requested by an Android application collide with the user's privacy assumptions.

8 Conclusion

The popularity of Apple's iOS and the AppStore attracted developers with malicious intents. Recent events have shown that malicious applications available from the AppStore are capable of breaching the user's privacy by stealing privacy sensitive information, such as phone numbers, address book contents, or GPS location data from the device. Although static analysis techniques have shown that they are capable of detecting such fraudulent applications, we are convinced that attackers will employ obfuscation techniques to thwart static analysis. Therefore, this paper discusses the challenges and open problems that have to be overcome to provide comprehensive dynamic analysis tools for iOS applications. We tackled two of these challenges by providing prototype implementations of techniques that are able to generate method call traces for iOS applications, as well as exercising application user interfaces. Our evaluation highlights the necessity for taking user interfaces into account when performing dynamic analysis for iOS applications.

Acknowledgements. This work was partially supported by the ONR under grant N000140911042 and by the National Science Foundation (NSF) under grants CNS-0845559, CNS-0905537, and CNS-0716095. We would also like to thank Yan Shoshitaishvili for his help with the evaluation device.

References

1. Apps - Android Market, `https://market.android.com/`
2. BigAdmin: DTrace,
   `http://www.oracle.com/technetwork/systems/dtrace/index.html`

3. iPhone Developer Program License Agreement, http://www.eff.org/files/20100302_iphone_dev_agr.pdf
4. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: Aeg: Automatic exploit generation. In: 17th Annual Network and Distributed System Security Symposium, NDSS 2011 (2011)
5. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: ASIACCS 2010: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, pp. 135–144. ACM, New York (2010)
6. Beschizza, R.: iPhone game dev accused of stealing players' phone numbers, http://www.boingboing.net/2009/11/05/iphone-game-dev-accu.html
7. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (August 2004)
8. Dinaburg, A., Royal, P., Sharif, M.I., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: ACM Conference on Computer and Communications Security (CCS), pp. 51–62 (2008)
9. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: 17th Annual Network and Distributed System Security Symposium, NDSS 2011 (2011)
10. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.X.: Dynamic spyware analysis. In: Proceedings of the 2007 USENIX Annual Technical Conference, pp. 233–246 (2007)
11. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware analysis techniques and tools. ACM Computing Surveys (to appear)
12. Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of OSDI 2010 (October 2010)
13. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 20th USENIX Security Symposium (August 2011)
14. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Security and Privacy 7(1), 50–57 (2009)
15. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A Survey of Mobile Malware in the Wild. In: ACM Workshop on Security and Privacy in Mobile Devices (SPSM), Chicago, IL, USA (October 2011)
16. B.R. for The Register. iphone app grabs your mobile number, http://www.theregister.co.uk/2009/09/30/iphone_security/
17. Hallaraker, O., Vigna, G.: Detecting malicious javascript code in mozilla. In: 10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), pp. 85–94 (2005)
18. Hunt, G., Brubacher, D.: Detours: binary interception of Win32 functions. In: 3rd USENIX Windows NT Symposium, pp. 135–143. USENIX Association, Berkeley (1999)
19. Mulliner, C., Vigna, G., Dagon, D., Lee, W.: Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 91–108. Springer, Heidelberg (2006)
20. Mutz, D., Valeur, F., Vigna, G., Krügel, C.: Anomalous system call detection. ACM Trans. Inf. Syst. Secur. 9(1), 61–93 (2006)

21. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: Proceedings of the 2006 EuroSys Conference, pp. 15–27 (2006)
22. Vasudevan, A., Yerraballi, R.: Stealth breakpoints. In: 21st Annual Computer Security Applications Conference (ACSAC), pp. 381–392 (2005)
23. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained malware analysis using stealth localized-executions. In: IEEE Symposium on Security and Privacy, pp. 264–279 (2006)
24. Vasudevan, A., Yerraballi, R.: Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In: Proceedings of the 29th Australasian Computer Science Conference, pp. 311–320 (2006)
25. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security and Privacy 5(2), 32–39 (2007)
26. Wired. Apple Approves, Pulls Flashlight App with Hidden Tethering Mode, `http://www.wired.com/gadgetlab/2010/07/apple-approves-pulls-flashlight%2dapp-with-hidden-tethering-mode/`
27. Yin, H., Song, D.X., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: ACM Conference on Computer and Communications Security (CCS), pp. 116–127 (2007)