Vision: Automated Security Validation of Mobile Apps at App Markets

Peter Gilbert
Duke University
Durham, NC
gilbert@cs.duke.edu

Byung-Gon Chun
Yahoo! Research
Santa Clara, CA
bgchun@gmail.com

Landon P. Cox
Duke University
Durham, NC
lpcox@cs.duke.edu

Jaeyeon Jung
University of Washington
Seattle, WA
jyjung@gmail.com

ABSTRACT

Smartphones and "app" markets are raising concerns about how third-party applications may misuse or improperly handle users' privacy-sensitive data. Fortunately, unlike in the PC world, we have a unique opportunity to improve the security of mobile applications thanks to the centralized nature of app distribution through popular app markets. Thorough validation of apps applied as part of the app market admission process has the potential to significantly enhance mobile device security. In this paper, we propose AppInspector, an automated security validation system that analyzes apps and generates reports of potential security and privacy violations. We describe our vision for making smartphone apps more secure through automated validation and outline key challenges such as detecting and analyzing security and privacy violations, ensuring thorough test coverage, and scaling to large numbers of apps.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Design, Security

1. INTRODUCTION

The success of Apple's App Store and Google's Android Market has transformed mobile phones into a first-class development platform. The number of third-party applications or *apps* that the average smartphone user installs has grown rapidly [6], and browsing app stores has become a form of inexpensive entertainment for millions of people. Apps are small programs that often provide their functionality by accessing sensitive data (e.g., account, password, contact, financial records, medical records, GPS, camera, and microphone) and services located in the cloud (e.g., Google, Facebook, and Twitter). Ensuring that apps properly handle such high-value sensitive data is an important and difficult problem.

MCS'11, June 28, 2011, Bethesda, Maryland, USA.

There are growing concerns about both buggy and malicious apps that may leak, steal, or destroy sensitive data. Recent incidences of malicious apps found in the Android Market show that smartphones are susceptible to the same kinds of malware that have long plagued the PC world [5]. Furthermore, many other apps, while lacking malicious intent, may unintentionally compromise sensitive data. Recent studies of the Android and iPhone platforms [8, 19, 27] have found that apps frequently share private data in an undesirable way by leaking it to unknown destinations and third-party ad servers.

Similar to approaches for securing PCs, research on mobile device security has explored end-host system solutions such as scanning for viruses using signatures. Our prior work on TaintDroid [19] used taint tracking to detect unwanted exfiltration of sensitive data. It allows a user to track the propagation of sensitive data through and between apps and can raise an alert when sensitive data leaves the device. However, by the time a leak has been detected and analyzed, it may be too late to protect the sensitive data. Other work explores running end-host replicas (dubbed virtual smartphones or clones) in the cloud to enable more powerful analysis [15, 25]. However, these approaches still rely on detecting malicious or abnormal behavior after apps have been installed and run on individual smartphones.

Fortunately, unlike in the PC world, we have a unique opportunity to improve the security of mobile applications thanks to the centralized nature of app distribution; users typically obtain apps through just a few popular app markets (e.g., Apple App Store, Google Android Market, Amazon's Appstore, Microsoft Windows Phone 7 Marketplace). Applying security validation at the app-market level offers a great opportunity to enhance mobile app security. However, app markets currently apply either limited manual validation or no validation at all. The manual validation approach involves experts employed by an app market or a third-party security firm deciding whether to "approve" an app by manually exercising its functionality and observing its behavior. Unfortunately, experience with Apple's App Store approval process has demonstrated that this approach is less than ideal. Apple's approval process can introduce costly delays and uncertainty into the development cycle, while banned behavior such as WiFi-3G bridging [9] and alleged violators of Apple's privacy policies [3, 4] have still slipped into the App Store.

We believe that automated validation of smartphone apps at the app-market level is a promising approach for drastically improving the security of smartphones. We envision an automated validation process applied either by market providers to apps submitted for in-

clusion, or by a third-party "market filter" service that advises users about apps' safety. Alternatively, a market provider could offload the task of validating submitted apps to a third-party service. In this paper, we present a system that aims to achieve this goal. At a high level, the system utilizes "virtual" smartphones running in the cloud to test and verify security properties of apps. By running virtual smartphones in parallel, we can analyze apps at a massive scale.

AppInspector is an automated security testing and validation system that embodies this approach. We have identified several important challenges such as generating inputs that sufficiently explore an app's functionality, logging relevant events at multiple levels of abstraction as the app executes, and using these logs to accurately characterize an app's behavior. Our exploration is preliminary and intended to initiate discussion on mobile app validation. The rest of this paper discusses each of these challenges in greater detail and proposes several promising techniques for addressing them.

2. SYSTEM OVERVIEW

We envision a security validation system that 1) analyzes apps submitted to popular app markets, 2) identifies apps that exhibit malicious behavior and should be removed or avoided by users, and 3) facilitates producing easy-to-understand reports informing users of potential privacy risks due to misuse or abuse of sensitive data.

To analyze apps, we propose a dynamic approach that monitors an app's use of sensitive information and checks for suspicious behavior such as excessive resource consumption or deleting user data. In order to scale to hundreds of thousands of apps in a cost-effective manner, this process must be automated as much as possible. However, it would be cost-prohibitive to test such a large number of apps on actual mobile devices. Instead, we propose using commodity cloud infrastructure to emulate smartphones. This will enable a large-scale security validation service to be built at low cost by utilizing the cloud for computation. A single host may be capable of running multiple "virtual" device instances at once, and cloud-hosted validation will enable testing many apps in parallel.

Building such an app validation system presents three key challenges:

- C1. How do we track and log sensitive information flows and actions to enable root cause analysis and application behavior profiling?
- C2. How do we identify security or privacy violations from collected logs and pinpoint the root cause and execution path that led to the violations?
- C3. How do we traverse diverse code paths to ensure that analysis is thorough?

In the rest of the paper, we give an overview of AppInspector, our proposed system to address these challenges. At a high level, the envisioned validation system consists of an AppInspector *master*, which creates multiple AppInspector *nodes*, each including a virtual smartphone. Validation is massively parallel, and requires little coordination between tasks. The master coordinates scheduling validation tasks on the nodes.

We outline the basic steps of a validation task, i.e., the analysis of an app on a single AppInspector node. Figure 1 illustrates the major components of the system mentioned in the overview. AppInspector first installs and loads the app on a virtual smartphone.

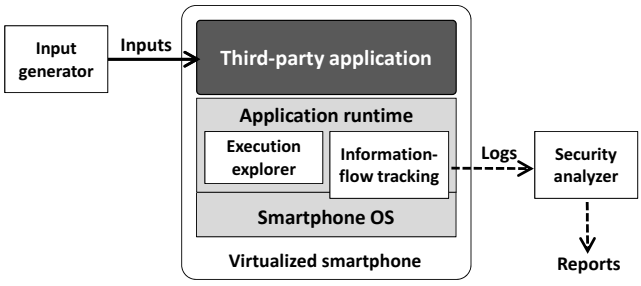

Figure 1: AppInspector node architecture

An *input generator* running on the host PC then injects user interface events and sensor input. The smartphone application runtime is augmented with an *execution explorer* that aids in traversing possible execution paths of the app. These two components address C3. While the app runs, an *information-flow and action tracking* component monitors privacy-sensitive information flows and generates logs, addressing C1. Finally, to address C2, AppInspector provides *security analysis tools* which can be used after execution completes to interpret the logs and generate a report.

In the following sections we further describe these four components.

3. TRACKING APPS' BEHAVIOR

AppInspector's ultimate goal is to identify third-party apps that behave maliciously or mishandle privacy-sensitive data, *before* they are released through app markets. An additional goal is to help smartphone users better understand how all apps handle their privacy-sensitive information, to allow them to make informed decisions about which apps to install and use. To this end, we must first define what we consider to be a security or privacy violation.

A **security violation** occurs when an app performs an action beyond the permissions granted to the app at install-time by the underlying smartphone platform. For example, if an app accesses sensitive data for which it is not granted permission, this is a clear security violation.

Privacy violations can be more subtle. Because many apps collect sensitive information such as location or user identifiers in order to provide useful functionality, simply detecting a transmission of sensitive data is not sufficient to declare a privacy violation. At a high level, a privacy violation occurs when an app releases sensitive data to a remote party in a way neither expected nor desired by the user. However, encoding user preference and expectations inside automated analysis is difficult. As a result, for the purposes of automated detection, we define a privacy violation as follows: we consider a violation to occur when an app discloses sensitive information without first notifying the user through a prompt or license agreement.

Whether or not a disclosure is considered a privacy violation by a user will often depend on its purpose or intent as perceived by the user: for example, a user may tolerate her location being sent to a content server to deliver content tailored to her location, but she might object to her location being sent to a third-party analytics service. In general, multiple components may be involved in causing a violation. These may include the app itself, as well as third-party analytics and advertising libraries plugged in by the developer for monetization. However, we note that the involvement of third-party code is not necessary for a violation to occur.

The key asset that AppInspector aims to protect is users' privacy-sensitive data. In order to detect leaks or disclosures and then iden-

tify the specific functionality or code component(s) involved in a leak or disclosure, we need to pinpoint the root cause and execution path that led to an outgoing network transmission containing sensitive data. To support this kind of analysis, it is necessary to track both *explicit flows*, in which sensitive information propagates through the app, external libraries, and system components through direct data dependencies, as well as *implicit flows*, e.g., when information is leaked due to a sensitive value influencing the control flow of the program.

Tracking Explicit Flows. To track explicit flows of sensitive data, we apply system-wide *dynamic taint analysis*, or taint tracking [17, 23]. Taint tracking involves attaching a "label" to data at a sensitive *source*, such as an API call which returns location data, and propagating this label through program variables, IPC messages, and persistent storage, to detect when it reaches a *sink* such as an outgoing network transmission. We take advantage of the fact that apps are often written primarily in interpreted code and executed by a virtual machine to simplify the implementation and reduce the runtime overhead of taint propagation as in TaintDroid [19]. However, since our analysis is performed offline, we can take a step further to address some limitations of the system posed by its real-time operation. For example, we can perform finer-grained taint tracking for IPC messages and file I/O to avoid overtainting problems. In addition, we can also explore finer-grained taint tracking for native functions.

Tracking Implicit Flows. Implicit flows leak sensitive information through program control flow. For example, consider the following if-else statement: `if (w == 0) x = y; else z = y;` where the value of `w` is privacy-sensitive. By watching the values of `x` and `z`, which are affected by the control flow, one can learn whether `w` is 0 or not. To detect such leaks via implicit flows, we can track control dependencies by creating control-dependency edges; e.g., in the above example, edges between `w` and `x` and between `w` and `z`. This can potentially result in *overtainting*, or labeling and propagating false dependencies. A possible approach for addressing this drawback is to selectively propagate tainted control dependencies as in DTA++ [20]. We note that tracking implicit flows accurately is a long-standing challenge and an active area of research.

Tracking Actions. In addition to flows of sensitive data, we track and log actions performed by applications at multiple levels in the software stack. For example, we record method invocations that lead to disk and network I/O events or access hardware devices such as GPS or accelerometers.

Choosing which information to log and the logging granularity is an important decision which affects both the depth and quality of analysis that can be performed later as well as the runtime performance of the app under testing. While it is not critical for a system driven by automated input to achieve real-time performance, large performance overheads could affect the number of execution paths that can be explored as well as the computational cost. With this in mind, we propose logging the following information: taint source and sink invocations, bytecode instructions which touch sensitive data along with code origin, call graph information including interpreted methods and native code (JNI) invocations, IPC and file system accesses involving sensitive data, and timestamps for all logged events. We believe that we can log these categories of information by instrumenting the application runtime and system libraries in a way that will not impose prohibitive performance or log volume overheads.

4. SECURITY ANALYSIS

The next challenge we consider is how to detect malicious behavior and misuse of sensitive data using our information-flow and action tracking runtime.

Dependency Graphs. An abstraction that we believe will prove useful is *dependency graphs*, which illustrate the path from the event determined to be the root cause of a malicious use or a misuse of sensitive data, through the data and control flow of the app and potentially other system components, to an eventual network transmission flagged as containing sensitive data. Dependency graphs are constructed once testing of an app completes using information collected during execution. On top of a dependency graph, we can perform analysis such as backward slicing, filtering, and aggregation. Backward slicing traverses vertices that are causally dependent from sinks to sources. Filtering produces a filtered log of an execution by excluding instructions which are unrelated and unaffected by sensitive information. Finally, aggregation produces a summarized log of an execution that affects sensitive information.

Using those primitives, we can write rules to identify misbehavior and then attempt to pinpoint responsible APIs (e.g., third-party library vs. application code), how data passed among apps (or between the app and the system), and the original action or API that triggered the misbehavior (e.g., platform API vs. a button click)

We expect that detecting security violations such as an app accessing sensitive data without acquiring explicit permission can be automated. However, it may be difficult to detect other misuses of sensitive data.

Detecting misuse of sensitive data. Detecting misuse of sensitive data by an app that was granted permission to access the data is more difficult. End-user license agreements (EULAs) and explicit notifications of data collection can provide useful hints for determining whether a disclosure of sensitive data amounts to a privacy violation. When a disclosure is detected, we can check if a notification is displayed to the user in the causal taint tracking path from the taint source to the taint sink. Then, we can try to determine if the notification contains messages informing the user of data collection or requesting permission to transmit the data in question. Similarly, we can check if the EULA mentions private data collection. Even with EULAs, some "permissions-hungry" apps may still request more permissions than they need to provide their functionality. Another facet of our analysis could be to determine the permissions actually needed by an app based on its observed functionality. This could help encourage developers to build more privacy-respecting apps.

Two promising approaches for interpreting the text of user notifications and EULAs are: 1) applying natural language processing and 2) crowdsourcing like Amazon Mechanical Turk [1]. In the future, this analysis could be made easier if developers utilize P3P [7] to express privacy policies in a machine-readable format.

5. INPUT GENERATION & EXECUTION EXPLORATION

So far, we have discussed how to track flows of privacy-sensitive information and analyze execution logs in order to identify security and privacy violations. A fundamental limitation of the proposed dynamic tracking techniques is that any analysis is limited to execution paths which are actually traversed during testing. In practice, an app may offer many diverse execution paths, and to ensure that privacy analysis is thorough, we must determine if there is a feasible path from any source of private information to a corresponding sink among all feasible execution paths. This poses another chal-

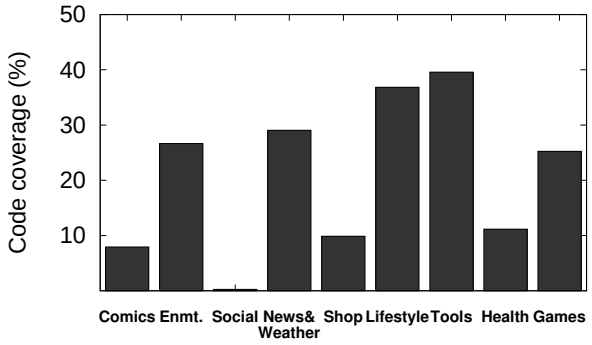

Figure 2: Code coverage for random testing (30 minutes)

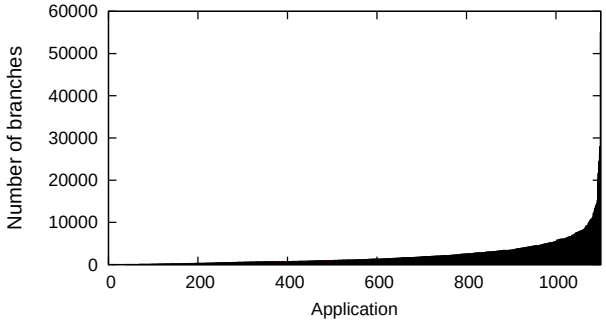

Figure 3: Branch counts of 1100 apps

lenge: *Can we explore diverse execution paths of an app accurately and scalably in an automated manner?*

First, we must keep in mind that smartphone apps are primarily event-based programs. Execution is typically driven by two broad types of events: 1) UI input events, such as taps or drags on the touchscreen or text inputs, and 2) callbacks triggered by other devices or sensors, such as GPS location updates and camera photos. As a result, our system must be capable of generating and delivering arbitrary sequences of UI and sensor input events. Fortunately, many apps use standard UI input elements such as text input fields and buttons provided by platform SDKs. To aid in delivering meaningful inputs to these apps, it may be useful to instrument standard UI libraries to treat common types of input elements such as username/password prompts as special cases.

When considering which execution paths to explore, we would like to avoid false negatives and false positives. A false negative occurs when testing fails to cover a feasible execution path that leads to potentially unwanted behavior such as transmitting sensitive data. On the other hand, a false positive occurs when testing explores and flags an execution path, when the path is globally infeasible. Exhaustively exploring all feasible execution paths, which yields no false negatives or false positives, is not scalable due to the well-known path explosion problem, i.e., the number of execution paths grows exponentially as the number of branches increases. This fundamental tradeoff between accuracy and scalability presents a number of research opportunities.

To begin, we consider the simple strategy of random testing, which explores concrete execution paths by injecting randomly-generated inputs, thus avoiding false positives and scalability problems. While attractive in its simplicity, random testing has been found to achieve poor code overage for other types of applications [10, 11]. To get an idea of whether random testing is a viable strategy for testing smartphone apps for privacy violations, we chose nine apps from a late 2010 survey of the most popular free apps in each category of the Android Market [2] and supplied each with a continuous stream of touchscreen taps and drags, hardware button presses, and location updates for 30 minutes. To measure coverage of execution paths, we modified Android's Dalvik VM to collect basic block code coverage. The results presented in Figure 2 show that random testing achieves 40% or lower coverage in all cases. While the experiments did yield at least two disclosures of location data, we observed that the tests commonly got "stuck" in terminal parts of the apps' UI. One example of the shortcomings of random testing is the test of a social networking app, which achieved less than 1% coverage because it could not progress past the initial login prompt. To enable more thorough exploration of execution paths, we advocate a more systematic approach.

Symbolic execution [21], which is typically used for finding bugs [11], systematically explores all possible execution paths. Applying symbolic execution to the entire software stack, including apps, libraries, and the OS, would eliminate all false negatives and false positives. However, as we mentioned earlier, complete system-wide symbolic execution is not scalable.

Instead, we propose applying a mixed-execution approach, also known as concolic execution, which combines symbolic and concrete execution (e.g., as in CUTE [26], EXE [12], and S2E [14]). Since our goal is to explore diverse paths of a specific third-party app, it is only necessary to apply symbolic execution to the app itself, while the rest of the environment (e.g., Android libraries and system) can be executed concretely. Selectively applying symbolic execution to only part of the system for scalability was explored by S2E [14]. However, for AppInspector, switching back and forth between symbolic execution and concrete execution is necessary not only for scalability but also to enable analysis of apps that communicate with remote parties. When an app interacts with a remote server hosted by a third party, it is not possible to symbolically execute the server code, so AppInspector must concretely execute request/response interactions with the server.

To test the basic feasibility of this approach in terms of scaling, we analyzed branch counts of 1100 apps, which are shown in Figure 3. 90% of the apps have 4187 or fewer branches. The results suggest that symbolic execution performed selectively may be feasible for these apps: a recent study [10] demonstrated path exploration for programs with similar complexity. Parallel symbolic execution [16] can further speed up symbolic execution.

In symbolic execution, we maintain state associated with the current execution path including a path constraint (a boolean formula over symbolic inputs) and symbolic values of program variables. When we switch from symbolic execution to concrete execution, we use a constraint solver to produce concrete values for the execution. To switch from concrete execution to symbolic execution, we can add the concrete return value and related side effects as part of a constraint. This can cause an overconstraining problem, which can in turn lead to false positives. Another possibility is to make the return value and related side effects symbolic; however this could cause the system to explore infeasible paths since we may not consider calling contexts properly. Exploring this tradeoff is an important research question. Finally, we must decide which program variables should be symbolic and which should be concrete. In general, we choose to make private information such as location symbolic; in contrast, for some variables such as remote IP addresses, we want to use the concrete value in order to determine where private data is being sent.

We are developing this mixed execution engine on top of the Android platform to evaluate the accuracy and scalability of our

proposed approach. At a high level, we plan to modify Android's Dalvik VM to add extra state including a path constraint or symbolic expression to local variables, operands, and fields. In addition, we must interpret bytecodes in a way that properly manages the extra state by updating the symbolic expression or by forking and updating state for a branch using a constraint solver.

6. RELATED WORK

We briefly describe key related work on software security analysis. PiOS [18] shares the goal of investigating smartphone apps for potential privacy violations. Unlike our work, PiOS employs static data flow analysis techniques and is implemented for the Apple iOS system. The use of static analysis enables exploring broad execution paths including infeasible ones. However, it is prone to false positives because of well-known problems in static analysis such as alias and context sensitivity problems. Furthermore, it provides incomplete analysis due to the difficulty of resolving messaging destinations and handling system calls. We hope to overcome these challenges by directly instrumenting the smartphone platform and tracking information flow at runtime. Lastly, it is hard to detect certain malicious behavior such as deleting sensitive data using static analysis. By analyzing running programs, we can observe the behavior of apps and detect this type of malicious behavior.

Dynamic information flow analysis techniques have proven useful for intrusion detection and malware analysis. BackTracker keeps track of the causality of process-level events for backtracking intrusion [22]. Panorama uses the instruction-level dynamic taint analysis for detecting information exfiltration by malware in Windows OS [28]. Unlike these systems, we explore diverse execution paths systematically by mixing symbolic and concrete execution focusing on third-party smartphone apps.

Recently, TaaS proposed a service for automated software testing [13]. CloudAV proposed antivirus as an in-cloud network service [24]. Similarly, we envision AppInspector being used as a service for privacy validation of smartphone apps in the cloud.

7. CONCLUSION

This paper presents our vision for implementing automated security validation of mobile apps at app markets. We proposed AppInspector, a security validation service that analyzes smartphone apps submitted to app markets and generates reports that aid in identifying security and privacy risks. We sketched the high-level design of AppInspector and discussed several challenges and ideas for approaching them. We strongly believe that large-scale automated validation of apps at central distribution points is an important step toward enabling more secure mobile computing, and we urge the research community to take advantage of this opportunity.

8. REFERENCES

[1] Amazon mechanical turk. `www.mturk.com`.

[2] Android market. `market.android.com`.

[3] Apple sued over apps privacy issues; google may be next. `www.reuters.com/assets/print?aid=USTRE6BR1Y820101228`.

[4] iphone and android apps breach privacy. `www.foxnews.com/scitech/2010/12/18/apps-watching/`.

[5] Malware infects more than 50 android apps. `www.msnbc.msn.com/id/41867328/ns/technology_and_science-security/`.

[6] More than 60 apps have been downloaded for every iOS device sold. `http://www.asymco.com/2011/01/16/more-than-60-apps-have-been-downloaded-for-every-ios-device-sold/`.

[7] P3P 1.1 Specification. `http://www.w3.org/TR/P3P11/`.

[8] Your apps are watching you. `online.wsj.com/article/SB10001424052748704694004576020083703574602.html`.

[9] Flashlight app sneaks tethering into app store (for now) [pulled]. `www.macrumors.com`, July 2010.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *TR UCB/EECS-2008-123*, 2008.

[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *ACM CCS*, 2006.

[13] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *ACM SOCC*, 2010.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[15] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *EuroSys*, 2011.

[16] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In *LADIS*, 2009.

[17] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, 2007.

[18] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[20] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[21] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[22] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[23] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *NDSS*, 2005.

[24] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *USENIX Security*, 2008.

[25] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: Versatile protection for smartphones. In *ACSAC*, 2010.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *FSE*, 2005.

[27] E. Smith. iPhone applications & privacy issues: An analysis of application transmission of iPhone unique device identifiers (UDIDs). In *Technical Report*, 2010.

[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, 2007.