Exposing Software Security and Availability Risks For Commercial Mobile Devices*

Ryan Johnson, Zhaohui Wang, Angelos Stavrou, and Jeff Voas.

Key Words: Software reliability, Dynamic analysis, Execution coverage, Android

SUMMARY & CONCLUSIONS

The advent of smaller, faster, and always connected handheld devices along with the ever-increasing reliance on technology for our everyday activities have introduced novel threats and risks. Beyond hardware security another primary factor that affects the reliability of the device is mobile applications. Indeed, the shift to smart commercially available mobile devices has created a pressing need for understanding the risks in using third-party mobile code running on the mobile devices. This new generation of smart devices and systems, including iPhone and Google Android, are powerful enough to accomplish most of the user tasks previously requiring a personal computer. In our paper, we discuss the cyber threats that stem from these new smart device capabilities and the on-line application markets for mobile devices. These threats include malware, data exfiltration, exploitation through USB, and user and data tracking.

In this manuscript, we present our efforts towards a framework for exposing the functionality of a mobile application through a combination of static and dynamic program analysis that attempts to explore all available execution paths including libraries. We verified our approach by testing a large number of Android applications with our dynamic analysis framework to exhibit its functionality and viability. The framework allows complete automation of the execution process so that no user input is required. We also discuss how our static analysis output can be used to inform the execution of the dynamic analysis. Our approach can serve as an extensible basis to fulfill other useful purposes such as symbolic execution, program verification, interactive debugger, and other approaches that require deep inspection of an Android application.

In summary, we believe that our efforts are the beginning of a long journey to asserting and exposing the risks of commercially available mobile devices. Our future work will include non-Android platforms.

1 INTRODUCTION

Static analysis of application code serves as a useful method to examine the possible behavior that an application can exhibit; however, static analysis is constrained to certain functionality due to its inherent limitations of not actually executing the code [1]. Static analysis is susceptible to false positives, false negatives, and obfuscation [2, 3]. The precision of the analysis increases when the analysis process better understands the semantics of the code and is able to observe the state of an application. When using dynamic analysis, test inputs can be randomly generated, come from a pre-generated set, or be input by an active entity. Dynamic analysis may or may not get complete coverage of the code, but all the instructions executed will be reachable and the application's true behavior can be observed.

We have developed a process for static and dynamic analysis of Android programs. Our approach allows us to perform a quick, first-pass analysis and in-depth analysis to understand the behavior of Android applications. The dynamic analysis framework runs on a computer and performs concrete execution of an Android application while abstracting certain details from the execution of the application. This abstraction allows the dynamic analysis to automate the analysis of as many paths as possible through the application without requiring any user input. Due to the abstraction, automation is achieved, but the precision of the analysis is reduced. The abstraction is necessary due to not running the application on an Android-enabled phone and the absence of the Android Application Programming Interface (API) in disassembled applications, although we can still utilize the Java API calls resident within the Android API. The dynamic analysis framework only requires an Android Package (apk) file, which is the compressed format used to encapsulate the constituent files of an Android application into a single file.

To get as close as possible to complete coverage of the code, a method must exist to affect the control flow of the application. As each conditional statement is encountered, either the values of the variables would need to be changed at runtime to obtain the desired outcome, determined *a priori* by symbolic analysis, or be forced by controlling the jump to a particular branch independent of the outcome of the Boolean condition being evaluated. This type of execution approach [4, 5, 6] stresses the application by entering as many branches as possible to make the application exhibit different types of behavior.

The impetus behind this approach is to maximize the coverage in terms of code, as opposed to examining the behavior of the application exhibited by a more limited number of execution traces. This is important because malware can contain very specific conditions that must be met in order for it to display malicious behavior [7]. In certain instances, the behavior is

* Disclaimer: We identify certain software products in this document, but such identification does not imply recommendation by the US National Institute for Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.

triggered by certain events such as specific times, dates, hostnames, local IP addresses, the presence of a file, and other factors. In addition, an application may restrain its malicious functionality when it determines that it is being debugged, running in an emulator, or some other type of controlled execution environment [8].

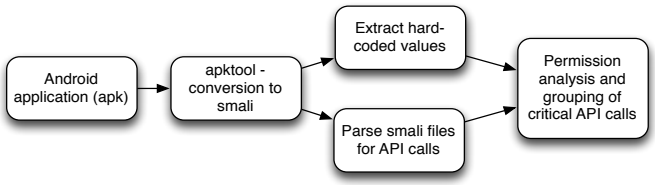

Figure 1. Overall task flow for Analysis of Android Applications.

2 STATIC ANALYSIS

To expose security and reliability risks, we developed a static analysis suite to quickly examine an Android application in order to examine the possible behavior of the application. Figure 1 shows the steps of the static analysis. The static analysis performs an in-depth scan of an application's requested permissions and its corresponding functionality based on the API calls detected in the decompiled code, and then reports any discrepancies between the two. Research has shown that Android application often ask for superfluous permissions for their applications [9]. In addition, an application may lack permission(s) even though its functionality justifies its inclusion in its AndroidManifest.xml file.

Certain Android API calls will require the application to declare one or more permission for the call to execute successfully. In addition, to the permission analysis that examines the correspondence between permissions and API call, the static analysis process also classifies sensitive API calls into categories based on behavior. After the static analysis has completed, a list of API calls, if any, corresponding to each category is generated. For the most part, these categories in the static analysis correspond to the groups that are used in the dynamic analysis. The groups are: commands executed, execution of binaries, Java reflection, loading of libraries, network events, files accessed, dynamic class-loading, etc. The list of API calls for each category also lists the file name and line number of each API call occurrence, so that further analysis can be performed to obtain more context for the API call.

The static analysis process enumerates the list of method calls and Android API calls that the application can make. The analysis also extracts hard-coded values from the application. It will parse through the smali [10] files of the application to find all the initialization of strings in the application. The smali format is a human-readable representation of the Dalvik bytecode, which shows the instructions as well as the register numbers, object types, and literal values that can be used as arguments to the instructions. This set of strings from the smali files can then be parsed for known malicious URLs or suspicious strings such as nc, su, etc. This only works for hard-coded values where the application developer has not made any attempt at obfuscating the value. For example, there is an Android application with the package name of *"com.antivirus.kav"*, which uses a simple technique to obfuscate the actual domain that it connects to. The method called LinkAntivirus in the application returns a domain that has been transformed using three calls to the replace method of the java.lang.String class. Figure 2 shows the smali for the LinkAntivirus method, which upon conclusion will ultimately return a String containing http://routingsms.com/z.php.

```
.method public static LinkAntivirus()Ljava/lang/String;
    const-string v0, "h=-q--=----tq--t-q=p-q=:-==q/q/qrqoqu-=t-i=qnq-gq=-
sqm=-sq.-=c-=qo-mq/=-qzq.-q=p=qh-p="
    const-string v1, "="
    const-string v2, ""
    invoke-virtual {v0, v1, v2}, Ljava/lang/String;->replace(Ljava/lang/
CharSequence;Ljava/lang/CharSequence;)Ljava/lang/String;
    move-result-object v0
    const-string v1, "-"
    const-string v2, ""
    invoke-virtual {v0, v1, v2}, Ljava/lang/String;->replace(Ljava/lang/
CharSequence;Ljava/lang/CharSequence;)Ljava/lang/String;
    move-result-object v0
    const-string v1, "q"
    const-string v2, ""
    invoke-virtual {v0, v1, v2}, Ljava/lang/String;->replace(Ljava/lang/
CharSequence;Ljava/lang/CharSequence;)Ljava/lang/String;
    move-result-object v0
    return-object v0
.end method
```

Figure 2. Simple domain name obfuscation using String manipulation.

We executed the application within the dynamic analysis framework to obtain a better understanding of the functionality of the application. The dynamic analysis output revealed that the application connects to this domain using the openConnection method of the java.net.URL class. The application also appends the phone number, the device ID, and the subscriber ID of the phone to this domain which occurs in the GetRequest method of the SmsReceiver class within the application. Performing static analysis on the files of the disassembled application would not be able to detect the obfuscated domain, as well as the phone-specific information it appends to the domain.

We plan to use the static analysis to identify all the sensitive calls in an application and then try to determine a path or paths to the sensitive call from one of the entry points of the application. The path information would be of the form of the application component to start from and then a sequence of conditional jump values to input, switch cases to enter, and callbacks to execute. A list of methods is not needed since the dynamic analysis program always executes each method call it encounters. The output of the static analysis can be used to guide the dynamic analysis as to how to reach the sensitive calls within an application. This would significantly improve the performance of the dynamic analysis since it currently performs an exhaustive search for un-traversed executed paths through the application. The static analysis output can

augment and reinforce the dynamic analysis process by directing it in an intelligent manner.

3 DYNAMIC ANALYSIS

3.1 Dalvik Bytecode and Java implementation

We utilized apktool, a free open-source utility, which unpacks an apk file into its constituent resources and disassembles the Android application's classes.dex file into a format called smali. Each application contains a single classes.dex file, and apktool disassembles it into a directory tree of smali files. Each smali file corresponds to a single Java class file. Android applications are written in Java and converted to Dalvik bytecode.

Dalvik bytecode uses 1 byte to denote the opcode for an instruction. This yields 256 possible instructions although there are currently only 226 instructions in use [11]. We examined the documentation for the instructions and developed a Java implementation for all of the instructions that occur in Dalvik bytecode. This enables us to perform concrete execution of an Android application's bytecode within certain limitations. Although the Android API is notably absent from the disassembled application, certain API calls from the wrapper classes for primitive data types, the java.lang.reflect package, the java.lang.String class, the java.lang.System class, the java.lang.Runtime class, and different classes to load classes dynamically are handled by our dynamic analysis framework. In addition, we also leverage the Java API since dynamic analysis executes inside the Java Virtual Machine, which allows us to creates a wrapper around certain Java API calls and have them executed.

Dalvik bytecode uses registers, as opposed to a stack, to pass and maintain the values of primitive data types and object references as an optimization for the mobile platform [12]. Our dynamic analysis creates a register entry object to keep track of the following data: type, object type (if applicable), value, register number, variable name, elements (for arrays), and fields (for objects). The value attribute in the register entry object is stored as a string, and it is converted to the format of its respective primitive data type once it is used in an operation. Dalvik bytecode does not declare the primitive data type belonging to the literal value being loaded into a register. Davlik bytecode has mathematical and logic instructions specifically to operate on different primitive data types. The value variable retains the literal value, in hex, in the register entry object until the register is used as a parameter to a mathematical or logic instruction. The type of the primitive data type can be inferred by the type of instruction that it is used in.

When an Android API call, which is not specifically handled by the dynamic analysis, is encountered, then the code for the API call is not actually executed and a register entry object of its return type, if any, is created. A default value is used for primitive data types in this scenario. We use an abstract representation for the objects that are returned from Android API calls, which are not specifically handled by our dynamic analysis. These objects will have the same corresponding type but they will have no state. The state can be built as operations from the application to set the value of the instance variables of an object.

This limitation can be overcome by incorporating a more dynamic approach where an Android-enabled phone is employed to handle all API calls that our dynamic analysis framework does not specifically handle. In addition, random values could be used for the values of the primitive data types, as well as the use of shadow objects. For a more dynamic approach, the phone would be tethered to the computer running the dynamic analysis. This could be achieved by using the standard format, which is used for the creation of objects and the calling of methods using Java reflection.

3.2 Dynamic Analysis - Structure and Operation

There are two primary components that comprise the dynamic analysis framework: the execution module and the controller module. The execution module handles the execution of the Dalvik bytecode and contains data structures related to a single execution path through the Android application. The controller contains a stack to temporarily retain data about each method that is called. As the execution encounters method calls and returns, the stack grows and shrinks, respectively. The controller module creates a new instance of the execution module after it completes each execution path. Certain data is transferred from the execution module to the controller module before each execution module is replaced by a new instance of itself. A single instance of the controller module persists until all the possible execution paths have been traversed or the time limit, if used, is reached. An optional time limit can be used to bound the execution time of the dynamic analysis. The execution of different paths through the application is modeled using a binary tree [13]. The controller module contains the logic to determine: (1) which path should be taken when a conditional statement is encountered, (2) the management of the binary tree (i.e., insertion of nodes and updating the state of nodes), and (3) various routines for detecting loops, maintenance of ancillary data structures, handling recursion, and detecting infinite loops.

An Android application can have various entry points into the application which are called application components [14]. The AndroidManifest.xml file enumerates the list of entry points into an application. The dynamic analysis framework parses the AndroidManifest.xml file to get the list of the application components and sequentially performs forced path execution on each. A binary tree is created for each application component. The dynamic analysis framework starts execution of the class constructor, constructor, and initial method (i.e., onCreate or onReceive) for the application component after examining its corresponding smali file which associates application components with their initial class. The initial method, and any other method that is called, is searched for infinite loops before the execution of the method begins.

In the case of a conditional statement or switch statement, the dynamic analysis will locate the appropriate branch to execute within the method by examining the binary tree to determine which path(s) from the node have not been traversed. In the event of a method call, the smali directory tree is searched to determine if the code is available in the

smali directory tree. If the code for the method is not available (i.e., Android API call) and the method has a return type that is not void, then a register entry object with no state is created with the return type of the method. If the code is present for the method call in the smali directory tree, all of the data related to the current method, variable values, execution location, and various data structures related to the processing of the method remain on the stack and the data related to the new method is pushed onto the stack. The stack easily enables the resumption of the execution of the previous method once a called method returns. The output from processing each application component is: (1) a method call graph, (2) control flow graph, (3) the output of an in-order traversal of the binary tree, (4) a list of the jump values for each conditional statement taken for each execution through the application component, and (5) a list of relevant behaviors of the application (e.g., commands executed, execution of binaries, Java reflection, loading of libraries, network events, files accessed, dynamic class-loading, etc.). The dynamic analysis process could easily be modified to search for and record any functionality or behavior.

3.3 Forced Path Execution

To get full coverage of the execution paths through the code, the dynamic analysis controls the outcome of the evaluation of the Boolean condition for conditional statements. When a conditional statement is encountered, the execution module queries the controller module for what the outcome of the Boolean condition should be depending on which path(s) from the node have not been taken. This alleviates the importance of actual values that are evaluated in conditional statements since all possible paths are taken (or as many as possible before time expires). Taking all available paths yields a comprehensive view of the application's behavior. Dalvik bytecode has 12 different instructions to evaluate **if** conditional statements and 2 instructions to evaluate **switch** statements. The 12 instructions for **if** conditional statements have two possible outcomes. If the Boolean expression in an **if** statement evaluates to true, then execution continues after jumping to a particular code branch. If the Boolean condition of an **if** statement evaluates to false, then execution continues on a different branch (i.e., linear execution from the conditional statement). **Switch** statements will have at least one case and possibly a default case that is executed if all of other cases evaluate to false. The **switch** case that evaluates to true will have execution continue at the specific branch associated with the **switch** case.

The paths taken through the application are modeled using a binary tree. The nodes in the binary tree represent conditional statements that alter the control flow of the application. As conditional statements are encountered they are inserted into the binary tree assuming that the conditional statements are not part of a path that has already been traversed. If execution traverses a path that partially overlaps with another path that has already been traversed, then the reference to the current node will be moved along already established nodes in the binary tree until the paths diverge. Specifically, a node can represent an **if** conditional statement or a case of a **switch** statement. Unconditional jumps are not included since they occur without any condition being evaluated. Each node in the tree contains the following components {type/name, relative path and file name, line number, method, and **switch** name (if any)}. These attributes of each node uniquely identify a conditional statement within an application.

Each node will have two child nodes unless it is a leaf node – leaf nodes represent the conclusion of an execution path through the application. This can be caused by the eventual return from the initial method, detection of an infinite loop, or encountering an API call that causes the JVM to exit. Once a conditional statement is encountered, the two options are to make the jump to a specific branch if the Boolean condition is true or to continue linear execution if the Boolean condition is false. In the binary tree, the right child represents the next **if** conditional statement or **switch** statement case that is encountered when the Boolean condition of the current conditional statement evaluates to true and the jump to that particular branch is taken. The left child represents the next **if** conditional statement or **switch** statement case that is encountered when the evaluated condition of the current conditional statement evaluates to false and linear execution occurs. During our dynamic analysis process, following a jump (i.e., Boolean condition is true) is represented by the integer value 1 and continuing linear execution (i.e., Boolean condition is false) is represented by the integer value of 0.

The binary tree contains instance variables that reference the current node, the previous node, and the root node. The current node variable represents the current conditional statement or **switch** statement case that is currently being processed. The previous node variable represents the node that was immediately processed before the current node. The root variable references the root node of the binary tree. Each node contains Boolean variables representing whether the node itself is finished, whether its left child node is finished, and whether its right child node is finished. A node is considered to be finished when its left child node and right child node are both finished indicating that all possible execution paths traversing through the node have been completed.

Once an execution path is finished, a finished node representing the completion of an execution path is inserted as the left child or right child of the current node depending on the value of the last jump taken. The finished node has its left child finished, right child finished, and node finished Boolean variables set to true. After the finished node is inserted, the tree is traversed to update any Boolean variables that indicate whether the Boolean variables for the left child, right child, or the node itself is finished for all nodes. This is a process that starts updating the Boolean variables after an execution path completes. The traversals occur until no Boolean values are modified throughout an entire traversal of the binary tree. As an execution path is taken, the Boolean variables representing whether a node's child nodes are finished for each node on the path are examined to ensure that the path has not already been traversed. This process repeats until the root node becomes finished by having both its left and right child nodes become finished which indicates that all possible paths through the application component has been traversed.

We have taken various measures to bound the execution time of an application during execution. We have limited recursion, cyclical calling of methods to each other, the number of loop iterations allowed, and execution time. We make an attempt to detect infinite loops, although it is not possible to detect all infinite loops due to the halting problem. There are certain cases where it is straightforward to detect an infinite loop such as a method lacking a return statement and continuous iteration of code between an unconditional jump and its target location that it jumps to without encountering a conditional statement. Bounding the analysis time of the application, however, obviates the possibility of execution causing an infinite loop.

3.4 Runtime File I/O Events

File I/O events can provide a valuable source of information for security inspection of Android applications [15]. The file I/O activities include file creation on the file system, reading from files, writing to them or deleting them from the file system. Malicious apps make extensive use of file I/O for various malicious activities. For instance, many malware families include publicly available root exploits in their packages in either plain or obfuscated form [16]. As another example, consider a malicious app which masquerades as a legitimate app to steal a user's credentials, the app might save the information as a file if a network connection is not currently available for sending the credentials back to its remote server. To avoid raising suspicion, a malicious app could delete its files once it has successfully used them for its malicious activities. To gain better visibility into file I/O activity of apps, we utilize system calls made to the kernel to fully capture the file I/O content.

The Linux kernel constitutes the lowest level in the Android's architecture. As a result, all user space requests made by apps have to pass through the system call interface to get executed in the hardware. Capturing the file I/O content at the system call level provides an accurate picture of app behavior. We use the *strace* program to capture a full dump (HEX and ASCII) of each *read* and *write* system call made on behalf of the app under monitoring. A numerical file descriptor specifies the target for a *read* or *write* system call. To gain better insight, we correlate the file descriptor argument of each *read/write* system call with its actual file. To do this, we record all the file descriptors to file mappings, which happen in the *open* system call. This way, we would be able to identify the file associated with a file descriptor used as an argument in a subsequent *read/write* system call. However, when the target of a *read* or *write* call is not a file on the file system, for instance, when named pipes or Unix domain sockets are used for inter-process communication, mapping will fail. To resolve such file descriptors, we look at the /proc/PID/fd path.

3.5 Runtime Network Communications

To capture the network traffic for all the applications, we run the application on a real device. By running applications on a real device, we overcome the inefficiency of the emulator. In addition, some malware may arm themselves with VM or emulator evasion techniques. The lack of cellular data network on an emulator may also affect the malware network footprint. The challenge for running applications on real devices is scalability. Even if we could have multiple devices to run different applications simultaneously on a "one app per device" basis, it is inefficient. To solve this problem, we run multiple apps in a single device simultaneously. However, network traffic capturing tools such as *tcpdump or wireshark* only can work at the device level. In other words, such tools cannot provide the granularity that is required by malware analysis to differentiate network connections on a per-app basis. As such, network address information is only available under the socket level given that we don't have access to the target malware application context at runtime. Therefore, we developed a dedicated kernel module to log the application and network address mapping in the kernel. Particularly, we monitor all *sys_connect* system calls and log the caller's process name and target network address information. For TCP/IP, we have the destination IP address and port number, while for the Unix domain socket we have the socket file name. By applying such mappings onto tcpdump capture file offline, we obtain the network traces of the malware application. With the storage space of the real device, we can run 200 applications in parallel within a single device.

4 RESULTS & FINDINGS

The GMU team tested the dynamic analysis framework on a number of malicious Android applications. Android applications can send text messages by declaring the SEND_SMS permission in its AndroidManifest.xml file and using API calls from the SmsManager class. We examined an Android application available online with the package name of com.ku6.android.videobrowser. This application is known-malware, which sends text messages to a premium phone number. We log the values of parameters to specific API calls that require security permissions or lead to administrative actions on the phone. For instance, we examined the sendTextMessage API call in the application which sends a text message to the destination phone number *"1066156686"*, a Chinese premium phone number and a text message body of the number 8. The application also transmits the phone's International Mobile Equipment Identity when connecting to the domain http://info.ku6.cn/clientRequest.htm. We have noticed similar functionality in an application with a package name of *"sectoolgoogle"*. This application sends a message of 1234567 to the destination number 10086. The application creates the directory /data/data/com.android.vending.sectool.v1/files/.hide/ to store various xml files. In addition, it connects to the following URL: http://www.youlubg.com:81/Coop/request3.php.

Android applications can also programmatically make phone calls without any user interaction by using the CALL_PHONE permission. We identified an application with the package name of com.xmedia.gobrowser that creates an Intent object containing URI with an Intent action of

android.intent.action.DIAL. The application then calls the startActivity method to send this Intent, which will call the number supplied as a URI. Upon investigation, this number appears to be registered to an operator outside US. The previous two aforementioned applications are well known. We were unable to find any analysis on the third application by searching for the phone number it contained or its package name.

5 LIMITATIONS

Our approach can be computationally expensive depending on the structure and size of the application being analyzed. Each **if** conditional statement that occurs outside of a loop exponentially raises the number of iterations that must be executed to cover all possible paths within an application. Loops that are deeply nested significantly affect the performance of the dynamic analysis due to the large number of iterations through the code, especially when many **if** conditional statements occur within each loop. An attacker could purposefully plant various computationally expensive activities throughout the application to slow the analysis of the application. There are approaches to make a trade-off between performance and analysis precision. There are the options to: (1) limit the number of iterations through a loop, (2) prevent loop nesting beyond a certain number of loops, (3) limit recursion, or (4) use a timer that sets a maximum time that can elapse to indicate that a path should end.

Currently, the dynamic analysis framework does not have support for multithreading. In addition, there are limitations to our approach due to the entropy in environment variables, user input, and non-deterministic routines: the analysis will enter all branches even if they are logically unreachable based on the set of inputs or environmental variables. The unreachable branches, i.e., conditional statements that will always be false, can result from programming logic errors. Upon execution of the application, the branch will never be entered when it is executed without forcing the outcome of a Boolean conditional associated with an **if** conditional statement. As the dynamic analysis encounters an **if** conditional statement, it will execute both branches of an **if** conditional statement without consideration as to whether the Boolean condition can never be true. This could result in a false positive when looking for certain behaviors if it occurs in an unreachable branch.

REFERENCES

1. M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, “Impeding Malware Analysis Using Conditional Code Obfuscation,” in Network and Distributed System Security Symposium (NDSS), 2008.
2. B. Chess and G. McGraw. Static analysis for security. IEEE Security and Privacy, 2(6):76-79, 2004.
3. A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In ACSAC, pages 421-430. IEEE Computer Society, 2007.
4. J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In Proceedings of the Symposium on Recent Advances in Intrusion Detection, 2007.
5. Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. Technical report CSE-2009-27, Department of Computer Science, UC Davis, 2009.
6. S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathEx- pander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro- architecture (MICRO), 2006.
7. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. Botnet Detection, 2008.
8. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland’07), May 2007.
9. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Proceedings of the 18th ACM conference on Computer and Communications Security, CCS 11, pages 627-638, New York, NY, USA, 2011.
10. smali - An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/.
11. Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html.
12. W. Enck, P. Gilbert, B. gon 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 Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation, pages 393–408, August 2010.
13. M. Neugschwandtner, P. M. Comparetti, and C. Platzer. Detecting malware's failover C&C strategies with squeeze. In Proceedings of the Annual Computer Security Applications Conference (ACSAC), pages 21-30, 2011.
14. Application Fundamentals|Android Developers. http://developer.android.com/guide/topics/fundamentals.htm.
15. Takamasa Isohara, Keisuke Takemori, Ayumu Kubota: Kernel-based Behavior Analysis for Android Malware Detection. CIS 2011, pp.1011-1015.
16. Yajin Zhou, Xuxian Jiang, Dissecting Android Malware: Characterization and Evolution. IEEE Symposium on Security and Privacy, 2012, pp.95-109.

BIOGRAPHIES

Ryan Johnson is a PhD. student at George Mason University.

Zhaohui Wang is a PhD. student at George Mason University.

Angelos Stavrou is an Associate Professor at George Mason University.

Jeff Voas is a Computer Scientist at NIST and was the President of the IEEE Reliability Society in 2003-2005 and 2009-2010.