Android Provenance: Diagnosing Device Disorders

Nathaniel Husted
Indiana University

Sharjeel Qureshi[*] Dawood Tariq Ashish Gehani
SRI International

Abstract

Mobile devices are a ubiquitous part of our daily lives. Smartphones are being used in many areas where data privacy and integrity are a concern. One threat to integrity and privacy is the existence of bugs in operating system code. Little has been done to provide tools for system-wide runtime profiling and accountability. We propose operating system auditing and data provenance tracking as mechanisms for generating useful traces of system activity and information flow on mobile devices. The goal of these traces is to enable debugging and profiling of complicated system issues such as increased power drain. We contribute a prototype system for Android-based mobile devices and provide realistic examples of how our system can be used for debugging.

1 Introduction

Mobile devices are now a ubiquitous part of our daily lives. Smartphones have now surpassed their feature phone brethren as the most used mobile devices [2]. Smartphone operating systems are very complicated. The level of complexity differs significantly from that of traditional feature phones. Smartphones share many engineering characteristics with their desktop operating system counterparts, but design foci have differed, given the form factor, memory, battery life, and processing power limitations. Unlike desktop operating systems where it is common to have many windows open at once, mobile devices focus on only one at a time. Many other processes exist in the background and can cause unforeseen problems as the user is not aware of their existence. For example, recent experimentation done in the Android "modding" community found that changing the random number generation from a blocking to a non-blocking process reduced user interface lag considerably, a concern that has traditionally plagued Android [4].

[*]Done while visiting SRI.

It is very difficult to find the root cause of complicated issues, such as blocking system calls that cause high user interface latency. Many elements, both apparent and others that seem unconnected, may be interacting to cause an issue. Finding the problematic elements can be challenging. Complicated bugs can affect data security, battery life, and system response time. In order to discover the cause of these problems, one must debug differing aspects from a wide span of the system. Traditional debugging makes this difficult as it is designed to provide very detailed information regarding one specific application. Debugging also radically modifies the execution environment, sometimes to the point where the bugs no longer manifest (a "Heisenbug"). A global view of the interactions between system processes is needed. Creation of the global view *must have a minimal performance impact* on the system. Provenance at the right level of abstraction is able to provide such a global view of the system in a meaningful manner. Provenance allows a developer to search the causal chain to find the functional elements that are possibly the source of a bug. While some have used provenance in the mobile space for security reasons [5], we are unaware of its use for mobile debugging. Chiarini, however, has used system provenance for server and workstation troubleshooting [3]. While we share some common goals with Chiarini, our operating environments differ considerably.

Our paper explains in detail how provenance can be used to help discover performance-related issues and bugs caused by the interaction of a wide array of mobile processes on one device. We contribute a prototype audit and provenance system running on the Android operating system. We provide a use case involving the Android platform, demonstrating the detection of aberrant application behavior that affects performance. We chose Android as it is the dominant operating system by market share in the United States [10]. As Android is built with the Linux kernel at its core, we adapted the Linux Audit [8] system for use on Android. We then adapted

the SPADE provenance tool from SRI [7] for use on the Android platform.

We provide background on Linux Audit and its port to Android in Section 2. Section 3 provides background on SPADE and how it interacts with auditing. Next, we provide a use case of provenance in a debugging and profiling capacity in Section 4. We conclude with a discussion of future work in Section 5.

2 Android System Audit

System-level auditing provides a rich source of information. A provenance system can then organize the immense amount of information into a coherent, structured form. This allows reasoning and analysis that are not possible with raw audit streams. As Android is a Linux-based platform, we make use of the standard Linux Audit system. The use of a standard system removes the duplication of effort of creating a loadable kernel module.

Linux Audit consists of both user space and kernel components. The kernel components include a central audit manager and hooks located in a number of kernel subsystems. These subsystems include the file system drivers, and SE Linux [12] and App Armor [11] access control. The kernel audit system also has an API for developers to add audit hooks for their own subsystems. The user space components exist to collect, process, control, and transmit audit records from the kernel. Audit is *far more efficient* at recording system events than user space software such as *strace* [1] because it limits the amount of context switching required between user space and the kernel. The audit user space components include a supporting set of plugins that allow the aggregation of audit data from many servers to one central auditing server. There is also a report generation system and an API for custom plugins to be developed.

The kernel components are able to record information about individual system calls, file system accesses, and SELinux policy events. Rules are used to match which events are of interest. Events can be selected based on their associated user identifier, group identifier, file name, or system call name. Other more complicated flags exist but were not used by us. The rules are configured by the `auditctl` utility in user space. The kernel events are sent to user space and handled by the `auditd` daemon. The `auditd` daemon will send events over a local socket to the `audispd` daemon, the audit dispatcher. The audit dispatcher interacts with the plugin system to disseminate records to connected applications.

Audit support was not available on the stock Android platform. We had to perform considerable work to enable system auditing. First, Android's kernel had to be modified to support system call auditing, which allows fine grain functional auditing at the operating system level. It was not supported on ARM, which is the architecture that nearly all Android devices run on. Our patch enabling system call auditing support for ARM has been accepted and is part of Linux kernels since version 3.3.

Adaptation of the user space components of audit to the Android platform was also non-trivial. One fundamental problem was that the audit user space utilities were written using the standard GNU *libc* C library. Android has only a nonstandard custom C library called "Bionic". We had to re-implement some fundamental C library functions such as `stpcpy`. Other design changes had to be made due to platform resource constraints.

Audit was originally designed to run on heavily used servers. Smartphones, when compared to servers, have considerably greater resource constraints in terms of memory, I/O, and processing power. To improve performance, we had to make a fundamental change to audit by removing the level of indirection from the dispatcher `audispd`. Previously, the audit daemon `auditd` would connect to the dispatcher through a local socket, which would then allow other applications to connect to it via a local socket. On servers, where processing speed and I/O are not an issue for audit, this socket configuration is not a problem. On smartphones, this leads to lost records when the local socket buffers fill up before they can be read. Removing the dispatcher eliminated one step in the chain of processing. Applications now connect directly with the audit daemon. This decreased the chance of lost or partial records. The resulting architecture is illustrated in Figure 1.

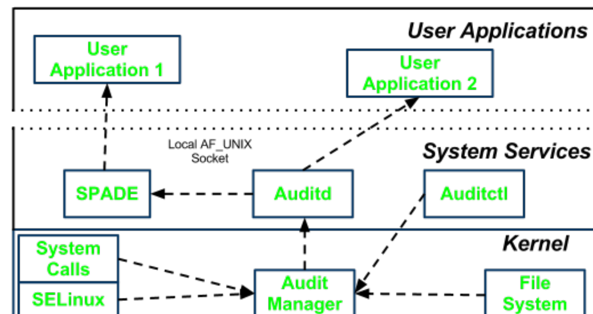

Figure 1: The architecture of our port of Linux Audit to Android, which is used by SPADE to generate data provenance.

Android also has a unique security scheme. Beyond basic POSIX permissions (read, write, execute) for files, Android implements a more abstract permission scheme. In order to secure access to audit's local socket connection (which retrieves records from an audit device), we needed to add a new permission to Android's infrastructure. This allows audit stream access to be limited to authorized applications. We had to modify Android's

startup configuration (since a system daemon was affected), Android's user space file system components, and Android's Java framework. Our "Audit for Android" release, including patches to enable Android permission-based access to audit records, can be found on Github [13].

3 SPADE on Android

We opted to adapt the SPADE [7] data provenance system to Android due to its modularity and portability. The core of SPADE is a provenance *kernel* that mediates interactions between provenance producers and consumers. The kernel's primary processing job involves buffering, filtering, and multiplexing the incoming data from multiple *reporters*, which are the event producers. The kernel then places the records into provenance storage that can be queried by provenance consumers. The underlying data structure in SPADE is a directed graph $G = (V,E)$, based on the Open Provenance Model (OPM). The types of vertices $v \in V$ are Processes, Artifacts, and Agents. The types of edges $e \in E$ are `used`, `wasGeneratedBy`, `wasTriggeredBy`, `wasDerivedFrom`, and `wasControlledBy`.

SPADE is written in Java and is thus easily cross compiled to Android's Dalvik virtual machine. We packaged it as an Android Application with permission to read the audit stream. SPADE was modified to adapt to Android's resource-constrained environment, which has less memory and processing power available. While SPADE on Linux is scheduled as a user space application, it does not use explicit synchronization with the audit stream. This means SPADE must read the audit stream fast enough to prevent buffers from being overwritten and records from being lost.

Due to limited resources on Android, garbage collection may be invoked frequently and may take more time than on desktops, particularly when memory pressure builds. SPADE's temporary buffers for parsing audit records allocate memory that must then be garbage collected, which halts the execution of applications when searching for unreachable memory references. This typically results in 30 to 70 milliseconds when SPADE is unable to process any events generated by Linux Audit. When the audit system generates a burst of records (in response to a spike in user activity), the frequent invocation of the garbage collection results in lower fidelity provenance (since records may be lost). We have addressed this by introducing a buffering utility written in C that uses explicit memory management. Its output is provided to SPADE via the Java Native Interface.

We needed to verify that the audit stream received from our patched system is accurate and that it results in correct provenance graphs. Since the accuracy of the system varies based on the load, simple unit tests reported correct operation even though the same code resulted in losses when running with a heavy workload. To analyze the accuracy of the system, we used synthetic workloads and logged data at each stage in the flow of audit records.

4 Provenance for Debugging and Profiling

With the increased complexity of systems comes difficulty when debugging and profiling applications. Much of the challenge manifests when analyzing complex system applications that can be affected by the behavior of apparently unrelated processes on the device. For example, a latency-critical application occasionally fails because another application has a lock on a file it needs. What the developer sees as a single function call to obtain a certain data element can turn into many inter-process communication (IPC) requests between many applications on the Android platform. Provenance provides a global view of the system, allowing us to observe when multiple applications access the same files or if a single function call translates to a large number of IPC calls.

We have examined two real-world use cases to illustrate the utility of provenance in the debugging and profiling contexts. The first involves provenance to detect *wake locks* on the Android platform. Wake locks are requests that applications make to the kernel in order to disable the system from entering a low-power sleep mode while work is being completed. The problem with wake locks is that they can have an extremely detrimental effect on battery life. While any one application might not use many wake locks directly, multiple wake locks may be used on its behalf through IPC calls that are hidden within the Android framework. An application inadequately relinquishing its wake locks can end up draining the battery on the device. OEM providers, custom ROM builders, and application developers are particularly interested in tracing wake locks to provide better battery life for devices.

The second use case aims to find the cause of low application responsiveness. In some cases, the cause of lags can be traced to components that block on system calls. As was the case with wake locks, all the stakeholders involved are interested in tuning performance for a better user experience. These include the designers of the operating system, the hardware vendors who customize it to their platform, the developers of applications, and end users.

Methodology Performing the qualitative analysis for our use cases required the generation of provenance graphs. This was done with the Android port of SPADE. In addition to the audit records, SPADE used process-

and filesystem-related information from the `proc` and `sys` pseudo-filesystems. These subsystems also provided access to records of Android Binder (IPC) activity.

We used Samsung Galaxy Nexus phones for our tests. We patched the Android operating system to enable support for Linux Audit, and installed the SPADE Android application. For every test, we first started SPADE, then ran the test application, and stopped SPADE after an adequate period of data collection (which was usually a few minutes). To prevent SPADE from generating provenance about its own execution, we added extra rules for the audit process to filter out its own activity. We included monitoring of frequently used system calls, such as `read()` and `write()`. A detailed list of functions and the audit rules can be found in Appendix A. SPADE was configured to output provenance metadata in Graphviz format [14]. We exported the generated output from the device using the Android debugger tool `adb` that is shipped with the Android SDK. Subsequent analysis was performed on a more powerful desktop machine.

Initial visualization of the provenance was performed by converting the resulting Graphviz data into a format that could viewed with a web browser. The size of the output graph data depends on the length of the execution. In our test case, the graph had ∼900 vertices and ∼5000 edges. Since the amount of provenance metadata can grow very large, a mechanism to filter the graph was needed to be able to locate information of particular interest. For this purpose, we used SPADE's Graphviz Reporter on a desktop machine to replay the collected output, with the kernel configured to store the collected provenance in SPADE's graph storage. We could then use SPADE's interactive query client. Queries complete rapidly since they leverage the underlying Neo4j graph database. In particular, queries took ∼50 milliseconds for the graphs we examined.

Wake Locks We first identified a set of applications that would drain the battery quickly, and hypothesized that the cause of the battery drain was the overzealous use of wake locks. We monitored the programs with SPADE and used the following query to discover wake lock usage: `result = getEdges(location:*wake_*lock, null, operation:write)`. This query retrieves a set of edges along with their associated vertices. The first argument to `getEdges()` specifies that the source vertices should be wake lock and unlock artifacts. The second argument is `null`, indicating that any process is an acceptable match. The third argument limits the set of edges to ones where the process has performed a write operation that modifies the wake lock. The result showed `wake_lock` calls at six places in the provenance graph but only five calls to `wake_unlock`. This prevents the device from sleeping, causing the battery to drain. A wake

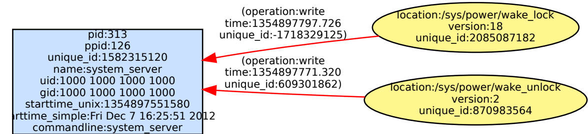

Figure 2: Correct behavior of a `wake_unlock` following a `wake_lock`.

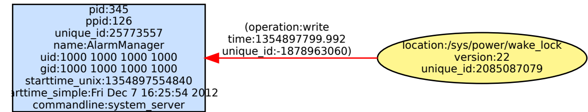

Figure 3: `AlarmManager` with a missing `wake_unlock` call causing increased battery drain.

lock and unlock pair can be seen in Figure 2.

We found that `AlarmManager` acquired the `wake_lock` but did not release it. `AlarmManager` is the Android component responsible for providing services for scheduling events in other applications. In this case, `AlarmManager` itself was being used by another application. Following the `AlarmManager`'s interactions, we were able to identify the application that was scheduling itself with `AlarmManager` but not releasing wake locks. The provenance trace for this behavior in `AlarmManager` can be seen in Figure 3.

Lag Reduction We studied the case of a software engineer who was improving a device's performance by replacing blocking calls to `/dev/random`. One alternative used calls to `/dev/urandom` while another option increased the entropy available to `/dev/random` [4]. The calls to `/dev/random` block if sufficient hardware entropy is not available, thereby causing significant delays. However, tracing all such calls along with their lineage is challenging due to the complexity of the system. We attempted to analyze the situation. In our experiment we ran SPADE for ten minutes and collected system-wide provenance. We then utilized the following query: `result = getEdges(null, location:/dev/*random, operation:read)`. This query identifies reads of `/dev/random` and `/dev/urandom` by any process, and returns the provenance edges with the incident vertices, so they can be examined further.

Performance Impact Monitoring low-level system functionality in real time can have a substantial adverse impact on performance. In order to determine the effect of using an Audit-enabled kernel and SPADE on system performance, we ran the AnTuTu 3.0.3 [9] Android benchmark application on a Samsung Galaxy

Nexus phone with 16 GB memory. AnTuTu runs various tests to measure the performance of CPU, memory, 2D and 3D graphics, SD card reads and writes, and database I/O. It assigns a score to the device. We ran benchmarks on three configurations: (1) a stock device with factory firmware that had not been modified, and (2) a device on which an Audit-enabled kernel was installed but the output records were not processed, and (3) a device with an Audit-enabled kernel and SPADE running. The results are shown in Figure 4.

Performance Benchmarks	
Configuration	AnTuTu Score
Factory Default	7890
Audit Only	7770
Audit with SPADE	7760

Figure 4: AnTuTu benchmark results showing negligible performance degradation from using SPADE and Audit on Android. For context, consider that benchmarking different phones can result in variations a thousand points [9].

The results show that there is little performance loss from enabling auditing and provenance collection. We also found that applications remain responsive and equally usable while auditing is active and provenance is being collected.

5 Conclusion and Future Work

Provenance has been relatively unused in the realm of mobile devices. Beyond that, provenance as a debugging and profiling tool has been little discussed in the community. We have provided a prototype system combining the Android operating system, the SPADE provenance middleware, and a port of the Linux Audit subsystem. The provenance from SPADE provided promising results for tracking down cases where wake locks were not being properly handled by applications and thus causing increased battery drain. We also showed that provenance can be used to trace the source of latency-inducing components. Through the use of the AnTuTu benchmarking suite, we were able to quantitatively show a negligible performance impact from using SPADE and Audit to collect system-wide provenance metadata. We plan to continue improving SPADE's performance on Android as well as increase the variety of information that can be obtained from the provenance records.

It is easy to see how such a debugging system might fit into a system developer's toolkit. Picture Alice sitting at her computer attempting to solve a bug in her company's custom security software for smartphones. The software is meant to provide high security guarantees but is also causing a massive power drain on the system. The normal debugging tools for Android were of no help as single-stepping through her code showed little information. She then runs our Audit / SPADE debugging tool with her system application to generate provenance data. As she knows it is a power problem, she focuses on wake locks and power-related system calls. This use case illustrates that the developer must still be knowledgeable about the system they work on in order to know what to look for with provenance queries. For example, they must know wake locks can greatly affect power usage or that certain system calls take considerably more processing power, and thus battery power, than others. However, the most time-consuming part of debugging can now be automated since all occurrences of the activity can be identified rapidly with suitable provenance queries. Alice's query results show that one of the APIs she uses is turning on a wake lock and never releasing it. Not only has she found the problem, she also has a fix that can be contributed back to the Android project.

There is still much work that can be done to expand the use of provenance on mobile platforms beyond its use for debugging. For example, being able to trace system activity and file accesses allows policy checking in a "bring your own devices" (BYOD) corporate environment where regulatory compliance is necessary. Unlike other methods involving taint tracking [6], we can perform coarser file-level data flow analysis with minimal performance overhead.

Provenance is a key technology to improve the stability and security of mobile devices. It can provide real-time information about meaningful system behavior without overburdening the operating system and causing a poor user experience. We believe that there is much work still to be done to integrate provenance into the mobile user experience.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant IIS-1116414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26, 2011.

[2] C. Burns. Nielsen: First time smartphone and feature phone usage equal. *http://www.slashgear.com/nielsen-first-time-smartphone-and-feature-phone-usage-equal-30220760/*, Mar 2012.

[3] M. Chiarini and S. Harvard. Provenance for system troubleshooting. In *25th USENIX Conference on Large Installation System Administration*, Dec 2012.

[4] XDA Developers. Seeder 1.2.6 entropy generator to provide significant lag reduction. *http://forum.xda-developers.com/showthread.php?t=1987032\&nocache=0*, Nov 2012.

[5] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, 2011.

[6] 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 real-time privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *13th ACM/IFIP/USENIX International Conference on Middleware*, 2012.

[8] Steve Grubb. Linux audit. *http://people.redhat.com/sgrubb/audit/*, Jan 2013.

[9] AnTuTu Hong Kong. Antutu benchmark. *http://www.antutu.com/*, Jan 2013.

[10] D. Neal. Google's Android OS has over half of the US smartphone market. *http://www.theinquirer.net/inquirer/news/2202950/google-s-android-os-has-over-half-of-the-us-smartphone-market*, Sep 2012.

[11] AppArmor Project. Apparmor wiki. *http://wiki.apparmor.net/index.php/Main_Page*, Jan 2013.

[12] SELinux Project. Selinux wiki. *http://selinuxproject.org/page/Main_Page*, Jan 2013.

[13] https://github.com/nwhusted/AuditdAndroid.

[14] http://www.graphviz.org/.

A Appendix: Audit Rules

The auditctl application that communicates with the kernel's audit subsystem has a specific syntax for configuration rules. These rules can be either defined in an audit.rules file or via the command line. In our system, SPADE executes auditctl with the command line specified in Figure 5. SPADE programmatically determines the process identifiers of the Android debugger, all audit-related daemons, and SPADE itself. It then notifies the audit subsystem that it should ignore these processes. If these processes are not ignored, programs such as adb can create read and write loops that result in poor system performance and capture useless information.

```
rules := "auditctl -a exit, always
-S read -S readv -S write -S writev
-S link -S symlink -S mknod
-S rename -S dup -S dup2 -S setreuid
-S setresuid -S setuid -S setreuid32
-S setresuid32 -S setuid32 -S chmod
-S fchmod -S pipe -S connect
-S accept -S sendto -S sendmsg
-S recvfrom -S recvmsg -S pread64
-S pwrite64 -S truncate -S ftruncate
-S pipe2 -F success=1 |ignorePids|"
```

Figure 5: The auditctl command line utility is executed by SPADE to configure the audit rules. Arguments following a -S are system calls. We only track system calls that return successfully. |ignorePids| is a list of process identifiers that refer to audit, the Android debugger *adb*, and SPADE.

B Appendix: Global Provenance Example

The enormous size of a global provenance graph for a system prevents it from being published in the analysis section of our work. Instead, we provide an image of what part of a system-wide provenance graph from SPADE looks like. Such a graph can be seen in Figure 6. The large amount of data that SPADE collects is apparent. However, to extract useful information from that data, it must be queried and analyzed. SPADE provides a mechanism to do this on the device, but querying was done on an external platform to reduce processing time.

Figure 6: A portion of a provenance graph for the Android system.