Better Performance Through Thread-local Emulation

Ali Razeen, Valentin Pistol, Alexander Meijer, Landon P. Cox
Duke University

ABSTRACT

Mobile platforms are shifting away from managed code and toward native code. For example, the most recent versions of Android compile Dalvik bytecodes to native code at install-time, and apps frequently use third-party native libraries. The trend toward native code on mobile platforms calls us to develop new ways of building dynamic taint-tracking tools, such as TaintDroid, that achieve good performance. In this paper, we argue that the key to good performance is to track only when necessary, e.g., when an app handles sensitive data. We argue that *thread-local emulation* is a feature that captures this goal. In this paper, we discuss the motivation for thread-local emulation, the software and hardware techniques that may be used to implement it, results from preliminary work, and the many challenges that remain.

1. INTRODUCTION

Dynamic information-flow analysis (i.e., *taint-tracking*) underlies a wide range of experimental mobile services, including secure deletion [19], protecting user privacy [9], and attesting to data authenticity [11]. TaintDroid [9] is the most widely used implementation of taint-tracking for mobile platforms, and it owes much of its success to good performance.

TaintDroid tracks tainted data by interposing on bytecodes within the Dalvik virtual machine, which is a managed-code runtime like the Java Virtual Machine (JVM) or Common Language Runtime (CLR). Because managed code typically runs more slowly than native code, integrating taint-tracking logic into the Dalvik virtual machine introduces little additional slowdown. However, not only do apps increasingly rely on their own fast native code, but with the introduction of the Android Runtime (ART), Android now compiles developer-supplied Dalvik bytecodes into native code at install time [1]. Both trends have rendered TaintDroid's approach to tracking obsolete, and raise an important question: is it possible to build a practical implementation of taint tracking for mobile platforms dominated by native code?

HotMobile '16, February 23–24, 2016, St. Augustine, FL, USA

DOI: http://dx.doi.org/10.1145/2873587.2873601

Prior work on taint tracking native code is not promising. These systems typically deliver prohibitive slowdowns of between 10 and 30x compared to untracked native code [7, 15]. However, we observe that apps may not spend much of their time handling tainted data. For example, a blogging app may handle a user's password only while authenticating and not after. Similarly, relative to the time spent pulling and displaying friends' content, a social media app may spend little of its time processing and uploading images taken by the device's camera. Thus, if one were only interested in tracking passwords or images taken with the camera, then perhaps the system could taint track only during the initial login process or when the camera is used. By taint tracking *selectively*, the performance penalty may be made proportional to the number of instructions that handle tracked data.

Nearly a decade ago, the Xen team described a trap-and-emulate approach to selective taint tracking that uses page protections to identify when a program accesses tainted data [13]. Unfortunately, relying on page protections alone to implement selective taint tracking is insufficient on mobile platforms. Mobile apps are inherently multi-threaded and run on devices with multi-core CPUs. Since all threads in a process must adhere to the same page protections, when any thread handles tainted data, all of the app's active threads must also be emulated.

Thus, for modern mobile systems, we believe that practical native taint tracking requires *thread-local emulation*. Thread-local emulation allows threads that do not handle tainted data to run at full speed, while slowing down only those threads that handle tainted data. Thread-local emulation is particularly important for mobile apps, in which the primary UI thread must not be slowed by taint-tracked background threads. In this paper, we argue that thread-local emulation would offer significant benefits to mobile systems, and we explore some potential ways to implement it.

The rest of this paper is organized as follows. In Section 2, we provide background on taint tracking. In Section 3, we describe why a simple trap-and-emulate approach to selective taint tracking is insufficient for Android. In Section 4, we describe several ways to implement thread-local emulation. In Section 5, we present results from our preliminary work. In Section 6, we speculate about how to integrate taint tracking into Android without relying on the Dalvik VM. Finally, in Section 7 we present related work, and in Section 8, we provide our conclusions.

2. BACKGROUND: TAINT TRACKING

Taint tracking records data dependencies between program storage, such as individual memory addresses and registers, and one or more taint sources, such as a device's GPS sensor or the network. Each storage location has an associated taint label indicating whether its current value depends on a taint source. Because taint labels are not part of the physical- or virtual-machine interface, a taint-tracking system must update labels through *emulation*. That is, the system must interpose on any operation that transfers information from one storage location to another, such as a virtual-machine bytecode or physical-machine instruction. For example, TaintDroid maintains a 32-bit label for each object field and program register and updates fields' and registers' labels whenever a Dalvik bytecode executes.

For many years taint tracking was most commonly used to detect and defeat malware [15, 23], but many recent systems demonstrate that it is a generally useful technique for addressing a range of problems. For example, YouProve [11] uses taint tracking to attest to the authenticity of sensor data generated by a smartphone, such as audio or images, even if the data is compressed or cropped. Pebbles [19] uses taint tracking to logically group data items that are scattered across disparate parts of a storage system into higher-level units, such as an email or expense report. Grouping related data items into logical units allows the system to securely delete sensitive emails and other documents that span multiple databases and files. CleanOS [20] uses tracking to record where sensitive data resides in a device's memory and filesystem, so that this data can be evicted to the cloud if the device is lost or stolen. Finally, taint tracking has helped monitor how apps use private data [9], verify the security of point-of-sale apps [10], and improve apps' energy efficiency [18].

For nearly all of these systems, efficient taint tracking is crucial. The extra work required by emulation, i.e., interposing on each operation to update taint labels, imposes an inherent performance penalty. However, if the additional work is small relative to the time required to perform the original operation, then the overall performance penalty will also be small. This is precisely how TaintDroid achieves good performance; the Dalvik bytecodes on which it interposes are slow compared to native ARM instructions. At the same time, the poor performance of Dalvik bytecodes has led app developers to use more native code, and recent versions of Android compile bytecodes to native instructions at install time. This move to native code has had the unfortunate side effect of inflating the cost of taint tracking. Taint tracking native code typically costs between 10 and 30x, whereas TaintDroid's overhead is less than 20%.

One way to make taint tracking practical again is to do it selectively. In particular, apps may only rarely handle tainted data, and thus it may be possible to pay the price of emulation only when required. For example, if a system is only interested in tracking data from the GPS sensor, then apps that do not use location data will not pay a performance penalty. Or if an app uses location data, then it will only slow down while processing location data. Of course, for apps that make heavy use of tracked data, then the slowdown will be severe and unavoidable. However, we expect that many interesting apps will spend little time, relative to their lifetimes, handling tracked data. A major part of our ongoing work is testing this hypothesis.

For the rest of this paper, we concern ourselves with only explicit information flows, and we ignore implicit flows, e.g., information flows triggered by a program's control flow. The impact of this limitation depends on the use case. Prior work on implicit flows has shown that for certain applications and data types, implicit flows can be tracked at a reasonable cost. For example, SpanDex [8] quantifies and limits the amount of password information leaked through a malicious app's control flow. A general solution to the problem of implicit flows is unlikely to emerge anytime soon, but tracking implicit flows is unnecessary for many systems. For example, in YouProve, an app using implicit flows to leak information does not undermine the system's goals. YouProve will not attest to the authenticity of sensor data leaked through implicit flows, which defeats the reason for using it in the first place.

3. SELECTIVE TAINT TRACKING

To begin, we will assume that a mobile app consists of Dalvik bytecodes executing within a Dalvik VM and native libraries executing on the bare metal. TaintDroid provides taint tracking for bytecodes, but does not handle native code.

To implement selective taint tracking of native code, we must capture transitions from Dalvik to native code. Android apps are written in Java and use the Java Native Interface (JNI) to invoke native methods. Thus, we must first modify the JNI bridge and use the `mprotect` system call to disallow read and write access to memory pages containing tainted data when a native method is called from Dalvik.

As long as native code does not access tainted data, it will run at full speed. However, if native code tries to access tainted data, the operating system will raise a segmentation fault (`SIGSEGV`). The fault handler will transfer control of the thread to taint-tracking and emulation software running in the thread's address space. The emulator executes the program's instructions and performs dynamic taint tracking based on instructions' semantics. For example, suppose the emulator encounters the instruction `add r0, r1, r2`, which computes the sum of `r1` and `r2` and stores the result in `r0`. In addition to updating the value of `r0`, the emulator will update `r0`'s taint label using `r1`'s and `r2`'s labels: `Taint(r0)` $\leftarrow$ `Taint(r1)` $\cup$ `Taint(r2)`, where $\cup$ represents the union of two taint labels. The emulator executes native code in this manner until (1) it returns to Dalvik via the JNI bridge, or (2) all register labels are taint free[13].

While this approach works well for single-threaded programs, it creates a dilemma for apps that have threads executing in parallel. First, to emulate the faulting instruction, the fault handler must access tainted data. However, when the handler executes, it is still bound by the protections that caused the original fault. Accessing tainted data with these protections in place will trigger a new fault, leading to an unending series of faults and no forward progress. At the same time, allowing the fault handler to remove page protections to avoid an infinite loop would allow non-emulated threads to access tainted pages without trapping.

To prevent this, we could force all threads to run in emulated mode anytime a thread accessed tainted data. Placing each thread in emulation mode would allow us to safely disable page protections and rely on threads' emulation layers to protect tainted data. The downside of emulating all threads is that every thread would pay a significant perfor-

mance penalty, even when they do not access tainted data. This is particularly problematic for mobile platforms like Android, on which the main UI thread must remain fast to ensure good responsiveness. An ideal solution would only taint track the threads that access tainted data.

4. THREAD-LOCAL EMULATION

Thread-local emulation ensures that a thread will only pay the performance penalty of emulation if it accesses tainted data. Implementing thread-local emulation requires a way to protect tainted data from non-emulated threads while simultaneously granting emulated threads access. In this section, we discuss possible software and hardware techniques that may be used to do so.

4.1 Software Techniques

Reading Memory with the Kernel: Each process in the Linux kernel can access its own address space through the kernel by reading and writing `/proc/self/mem`. As the reads and writes take place in kernel space, they succeed without being restricted by page protections. Hence, the emulator can use it to access tainted data. Before emulating an instruction, it has to check if the instruction uses a memory location located in a protected page. If so, it has to perform the access via the kernel interface.

Multiple Virtual Page Mappings: The emulator can also use a technique proposed by Appel and Li [3]. For each page with tainted data, the emulator can create new entries in the process's page table that map to the same physical page, but with relaxed protections. Using these new mappings, the emulator can access tainted data without turning off page protections. Since the non-emulated threads will not be aware of the page table entries created by the emulator, they will not be able to freely access tainted data.

4.2 Hardware Techniques

We may also use hardware-specific features to implement thread-local emulation. Although this usually means a more complex implementation, hardware acceleration also gives us good performance. In this section, we focus on features provided in the ARM ISA. Although this limits the generality of the implementation, we consider it acceptable since most mobile devices use an ARM processor.

ARM Domains: The ARMv7 ISA has a feature known as memory domains. It allows each 1 MB region of a process's address space to be classified under different logical sections known as domains. There are a maximum of sixteen possible domains and each domain has one of three permissions associated with it. The domain permissions are set in a per-core CPU register known as the *domain control register*, which means that they are set on a per-thread basis. On an access to a memory location, the CPU core checks the running thread's permission for the domain of that location. If the permission is set to `CLIENT`, the core relies on page protections to decide if the access should be allowed. If it is set to `NONE` or `MANAGER`, access is unconditionally denied or allowed, respectively.

Memory domains may be used to implement thread-local emulation in the following manner. First, we define a new domain, `TAINT`. When a page contains tainted data, the surrounding memory region is classified under the `TAINT` domain. By default, a thread has `CLIENT` permissions for `TAINT`. The emulator, however, is given `MANAGER` permissions over `TAINT`. This means that even though the emulator runs in the same address space as the process, it can directly access tainted memory since its permissive `MANAGER` domain overrides page protections. Other threads will continue to be governed by page protections because they use the `CLIENT` permission.

Exposing Privileged CPU Features To User-Level Code: The inclusion of virtualization features in modern CPUs provides us with interesting opportunities. For instance, Belay et al. proposed Dune [5], a method of exposing privileged CPU features to user-level applications. In Dune, virtualization hardware features are used to allow applications to perform user-level page-table management. Applications can easily write their own page table entries and switch between different page tables. This is useful as our emulator can create its own page table mappings to freely access tainted data without also allowing non-emulated threads to do so. Unfortunately, Dune is not immediately usable in our work on smartphones for two reasons. First, Dune is implemented for the x86 ISA and not ARM. Second, as the authors of Dune admit, their page table management feature is not thread-safe. They state that this limitation is a matter of implementation but it is unclear how much work it would take to make it thread-safe in the context of ARM.

5. PRELIMINARY WORK

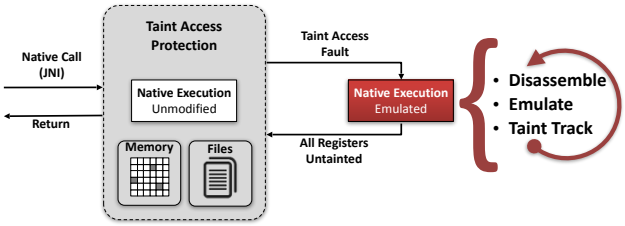

Figure 1: Overview of selective native code taint tracking with thread-local emulation.

In this section, we present results from our preliminary work on thread-local emulation for Android. A high-level overview of our system is illustrated in Figure 1. It follows a design similar to the one outlined in Section 3 and is currently implemented on Android 4.1.1.

Taint tracking at the bytecode level is done by TaintDroid. When an app uses the JNI, a taint protection layer ensures that memory pages containing tainted data are protected. If a thread attempts to access tainted data in native code, a taint access fault (a `SIGSEGV`) is triggered, which starts the emulator. The emulator disassembles each instruction, emulates its behavior, and performs taint propagation. When the CPU registers are no longer tainted, emulation ends and the thread executes natively.

To test the performance of our system, we developed a custom microbenchmark application, fully written in native code. This benchmark reads extended JPEG metadata (EXIF) from a tainted JPEG file using the `libjhead` native library and writes the metadata to standard out. It was developed based on our observations of the Instagram app. We found that it processes pictures taken with the camera in native code. During the processing step, it extracts the

EXIF data using the `libjhead` library and then applies a custom image filter.

We ran the benchmark on a Galaxy Nexus smartphone, and measured the time taken for it to complete, with and without taint tracking enabled. When taint tracking was enabled, we performed a full emulation of the benchmark and tried reading tainted memory with both the kernel and ARM domains. Our goal was to measure the amount of slowdown imposed by native-code taint tracking. Each experiment was conducted a hundred times. We report the median numbers below.

When the emulator reads tainted data through the kernel, the slowdown was 1,048x. In other words, the benchmark is over a thousand times slower when taint-tracking is enabled. As one might have predicted, trapping to kernel space each time we need a data item from a protected page is prohibitively expensive. When the emulator reads tainted data using ARM's memory domains, the slowdown is just 22x. We noticed similar levels of improvement in other data-heavy microbenchmarks.

Note that although a 22x slowdown is still significant, this is the slowdown of a microbenchmark that performs a single task. It does not reflect the amortized performance we expect with a third-party app that processes primarily untainted data.

5.1 Discussion

While implementing thread-local emulation, we noticed several issues that need to be addressed before we can have a practical system. In this section, we discuss each of them in detail.

Exiting Emulation Mode: We only want to emulate a thread while it is handling tainted data. Although it is safe to exit emulation when none of the CPU registers contain tainted data, it is not efficient to do so. Suppose we reach a state of untainted registers. Due to temporal locality, the next instruction may load tainted data again. Hence, it is wasteful to exit emulation as soon as the registers are untainted. We noticed this overhead due to unnecessary switching between native and emulation execution in our experiments. If we run our JPEG benchmark under selective taint tracking while using ARM domains, the slowdown is over 100x. Recall that the slowdown with full emulation is 22x. Our results are consistent with the observations made by Ho et al. in their selective taint-tracking work [13]. They mitigate this issue by continuing emulation for 50 more instructions after registers become untainted. Emulation exits only if the registers remain untainted at the end. We are presently investigating if a similar approach will suffice for us.

Tainted Data on Stack: The page containing the stack may become protected due to tainted registers that either spill onto the stack or are saved when a function call is made. A protected stack will immediately cause the thread to be emulated until the stack no longer contains tainted data. We are evaluating the performance impact of this issue and its relation to the temporal locality highlighted above. There are two ways of addressing this issue if it is a significant cause of overhead: (i) we could use a high number of watchpoints [12] if the hardware supports it to have fine-grained traps to tainted data, or (ii), we could use non-contiguous stacks [21] where tainted and untainted data are located in different pages even though they are on the stack.

False Positives: The page size on Android is 4 KB which means that page protections are coarse. Just as with the stack, a page with a single tainted byte will become protected and any access to untainted data within that page will cause unnecessary emulation. Although this is a general issue with page protections, it can take on special significance on Android. Android has a construct called the Looper [2], a message queuing system used in a multiple-producer, single-consumer manner. Each application has a single looper designated as the main looper. It is responsible for receiving events from different parts of the OS and delivering it to the application. If the page that contains the main looper's buffer becomes protected, the main event loop of the app will have to run in emulation mode. We are currently evaluating different options of addressing this issue, including the use of Dune [5].

The Trend to 64-bit ARM CPUs: We made use of memory domains, a feature of ARMv7, in our preliminary work. The ARMv7 ISA is used on 32-bit ARM CPUs. Unfortunately, the ARMv8 ISA, used on 64-bit ARM CPUs, no longer has support for memory domains. Given that newer smartphones are transitioning to 64-bit ARM CPUs [14], using memory domains is not a future-proof solution. This raises the urgency of using the other techniques discussed in Section 4 to allow the emulator to read tainted data.

6. LIFE WITHOUT DALVIK

In newer versions of Android, apps are still written in Java and compiled to Dalvik bytecode for distribution. However, the Dalvik VM is deprecated and is no longer used to interpret the app's bytecode. Instead, when an app is downloaded onto a phone, it is compiled into native code before execution. The absence of the Dalvik VM means TaintDroid will no longer work.

That said, the fact that apps are still distributed as bytecode provides us with an opportunity. We can rewrite its bytecodes and instrument them so that when the app is subsequently compiled, it will have taint tracking "baked in." This approach was first taken by Bell et al. in Phosphor [6], where tracking logic was added to Java applications running on the JVM by rewriting their bytecode. However, for performance reasons, we may not directly use their approach since in Android, the Dalvik bytecode will be further compiled into ARM assembly prior to execution. Instead of an always-on taint tracking scheme, as per Phosphor, we still need thread-local emulation.

To provide thread-local emulation we could first define a taint label for each class field. Second, we could set all class fields private and rewrite all direct field accesses as getter- and setter-method invocations. The getter for a field checks the field's taint label. If it is not tainted, then the field is returned. Otherwise, the getter raises a taint exception and passes the field's value in the exception object. This means that all getter invocations will be surrounded by a `try/catch` block. If the getter raises a taint exception, method execution continues in the catch block which contains the rest of the method together with the taint-propagation logic.

It is important to note that adding taint-tracking logic to the bytecode only replaces TaintDroid's functionality. It does not handle the cases in which apps may use native libraries via the JNI.

7. RELATED WORK

Taint tracking on mobile platforms is an ongoing research topic. Qian et al. proposed NDroid [17], a virtualized environment based on QEMU to run Android apps and track taints when they use native code libraries. NDroid relies on TaintDroid to perform taint propagation at the bytecode level. It uses software hooks implemented in QEMU to capture an app's use of JNI and run native taint tracking. Although NDroid performs native code taint tracking, it does not run on real smartphone devices, which is a key goal in our work.

In contrast to dynamic taint analysis, there has also been work on performing static taint analysis such as FlowDroid [4] and Amandroid [22]. The aim in both work is to detect malicious apps by performing static analysis on them and checking if they leak sensitive data. As stated in Section 2, we are interested in dynamic taint tracking as a general tool to solve a range of problems, instead of just malware detection. That said, these approaches complement ours. For example, static analysis may help during the byte code rewriting by highlighting the portions of the app where taint propagation logic is necessary.

Paupore et al. presented a preliminary design of a hybrid taint tracking system for mobile platforms in a recent paper [16]. They propose an always-on taint tracking system that runs with low overhead using a combination of static and dynamic taint analysis. The static analysis step runs first and adds special markers to paths in the app's managed code that uses sensitive data. When the app executes, the dynamic taint analysis system uses the markers to perform taint propagation. The dynamic taint tracker uses the *Embedded Trace Macrocell* (ETM), a hardware feature available on modern ARM processors, to improve performance. This feature allows a core to collect execution traces of a running thread and send it to another core, where it can be analyzed. The use of ETM allows app execution and taint propagation to run in parallel, as they are each done on separate cores. As with FlowDroid and Amandroid, the techniques used in this work complement our approach.

8. CONCLUSION

In this paper, we described *thread-local emulation*, a refinement of selective taint tracking. In thread-local emulation, only the threads that handle tainted data are emulated to perform dynamic taint tracking; all other threads run with minimal overhead. The goal of thread-local emulation is to make dynamic taint tracking practical in mobile systems dominated by native code by only performing taint tracking when necessary. Our preliminary work on implementing thread-local emulation on Android shows good promise. However, plenty of challenges lie ahead, especially achieving good performance on recent versions of Android.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Andrew Rice, for their insightful feedback. This work was partially funded by the National Science Foundation under awards CCF-1335443 and CNS-0747283.

10. REFERENCES

[1] Google I/O 2014 - The ART runtime. https://www.youtube.com/watch?v=EBlTzQsUoOw?t=37m25s.

[2] Android Developers Documentation. Looper. http://developer.android.com/reference/android/os/Looper.html.

[3] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of ASPLOS '91*, April 1991.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precice Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of PLDI '14*, June 2014.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of OSDI '12*, October 2012.

[6] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proceedings of OOPSLA '14*, October 2014.

[7] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of ISSTA '07*, July 2007.

[8] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. SpanDex: Secure Password Tracking for Android. In *Proceedings of USENIX Security '14*, August 2014.

[9] 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 *Proceedings of OSDI '10*, October 2010.

[10] W. Frisby, B. Moench, B. Recht, and T. Ristenpart. Security Analysis of Smartphone Point-of-Sale Systems. In *Proceedings of WOOT '12*, August 2012.

[11] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fidelity in Mobile Sensing . In *Proceedings of SenSys '11*, November 2011.

[12] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A Case for Unlimited Watchpoints. In *Proceedings of ASPLOS '12*, March 2012.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of EuroSys '06*, April 2006.

[14] Jerry Hildenbrand, AndroidCentral. Why 64-bit processors really matter for Android. http://www.androidcentral.com/why-64-bit-processors-really-matter-android.

[15] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of NDSS '05*, February 2005.

[16] J. Paupore, E. Fernandes, A. Prakash, S. Roy, and X. Ou. Practical Always-On Taint Tracking on Mobile Devices. In *Proceedings of HotOS '15*, May 2015.

[17] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On Tracking Information Flows through JNI in Android Applications. In *Proceedings of DSN '14*, June 2014.

[18] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing Mobile Apps To Use Sensor Hubs Without Programmer Effort. In *Proceedings of*

UbiComp '15, September 2015.

[19] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of OSDI '14*, October 2014.

[20] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. Clean OS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of OSDI '12*, August 2012.

[21] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of SOSP '03*, October 2003.

[22] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of CCS '14*, November 2014.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of CCS '07*, October 2007.