DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis

Lok Kwong Yan[†‡] Heng Yin[†]

[†]*Syracuse University*
Syracuse, New York, USA

[‡]*Air Force Research Laboratory*
Rome, New York, USA

{*loyan, heyin*}*@syr.edu*

Abstract

The prevalence of mobile platforms, the large market share of Android, plus the openness of the Android Market makes it a hot target for malware attacks. Once a malware sample has been identified, it is critical to quickly reveal its malicious intent and inner workings. In this paper we present DroidScope, an Android analysis platform that continues the tradition of virtualization-based malware analysis. Unlike current desktop malware analysis platforms, DroidScope reconstructs both the OS-level and Java-level semantics simultaneously and seamlessly. To facilitate custom analysis, DroidScope exports three tiered APIs that mirror the three levels of an Android device: hardware, OS and Dalvik Virtual Machine. On top of DroidScope, we further developed several analysis tools to collect detailed native and Dalvik instruction traces, profile API-level activity, and track information leakage through both the Java and native components using taint analysis. These tools have proven to be effective in analyzing real world malware samples and incur reasonably low performance overheads.

1 Introduction

Android is a popular mobile operating system that is installed in millions of devices and accounted for more than 50% of all smartphone sales in the third quarter of 2011 [22]. The popularity of Android and the open nature of its application marketplace makes it a prime target for attackers. Malware authors can freely upload malicious applications to the Android Market[1] waiting for unsuspecting users to download and install them. Additionally, numerous third-party alternative marketplaces make delivering malicious applications even easier. Indeed recent research has shown that malicious applications exist in both the official and unofficial marketplaces with a rate of 0.02% and 0.2% respectively [41].

[1]The Android Market has been superceded by the Android Apps Store in Google Play.

Malware analysis and exploit diagnosis on desktop systems is well researched. It is widely accepted that dynamic analysis is indispensable, because malware is often heavily obfuscated to thwart static analysis. Furthermore, runtime information is often needed for exploit diagnosis. In particular, much work has leveraged virtualization techniques, either whole-system software emulation or hardware virtualization, to introspect and analyze illicit activities within the virtual machine [11, 15, 18, 31, 33, 39, 37].

The advantages of virtualization-based analysis approaches are two-fold: 1) as the analysis runs underneath the entire virtual machine, it is able to analyze even the most privileged attacks in the kernel; and 2) as the analysis is performed externally, it becomes very difficult for an attack within the virtual machine to disrupt the analysis. The downside, however, is the loss of semantic contextual information when the analysis component is moved out of the box. To reconstruct the semantic knowledge, virtual machine introspection (VMI) is needed to intercept certain kernel events and parse kernel data structures [16, 21, 24]. Based on this idea, several analysis platforms (such as Anubis [1], Ether [15], and TEMU [35]) have been implemented.

Despite the fact that Android is based on Linux, it is not straightforward to take the same desktop analysis approach for Android malware. There are two levels of semantic information that must be rebuilt. In the lower level, Android is a Linux operating system where each Android application (or App in short) is encapsulated into a process. Within each App, a virtual machine (known as the Dalvik Virtual Machine) provides a runtime environment for the App's Java components.

In essence, to enable the virtualization-based analysis approach for Android malware analysis, we need to reconstruct semantic knowledge at two levels: 1) OS-level semantics that understand the activities of the malware process and its native components; and 2) Java-level semantics that comprehend the behaviors in the Java com-

ponents. Ideally, to capture the interactions between Java and native components, we need a unified analysis platform that can simultaneously rebuild these two semantic views and seamlessly bind these two views with the execution context.

With this goal in mind, we designed and implemented a new analysis platform, *DroidScope*, for Android malware analysis. DroidScope is built on top of QEMU (a CPU emulator [3]) and is able to reconstruct the OS-level and Java-level semantic views completely from the outside. Enriched with the semantic knowledge, DroidScope further provides a set of APIs to help analysts implement custom analysis plugins. To demonstrate the capability of DroidScope, we have implemented several tools, including native instruction tracer and Dalvik instruction tracer to obtain detailed instruction traces, API tracer to log an App's interactions with the Android system, and taint tracker to analyze information leakage.

We evaluated the performance impacts of these tools on 12 different benchmarks and found that the instrumentation overhead is reasonably low and taint analysis performance (from 11 to 34 times slowdown) is comparable with other taint analysis systems. We further evaluated the capability of these tools using two real world Android malware samples: DroidKungFu and DroidDream. They both have Java and native components as well as payloads that try to exploit known vulnerabilities. We were able to analyze their behavior without any changes to the virtual Android device, and obtain valuable insights.

In summary, this paper makes the following contributions:

- We describe two-level virtual machine introspection to rebuild the Linux and Dalvik contexts of virtual Android devices. Dalvik introspection also includes a technique to dynamically disable Dalvik Just-In-Time compilation.
- We present DroidScope, a new emulation based Android malware analysis engine that can be used to analyze the Java and native components of Android Applications. DroidScope exposes an event-based analysis interface with three sets of APIs that correspond to the three different abstraction levels of an Android Device, hardware, Linux and Dalvik.
- We developed four analysis tools on DroidScope. The *native instruction tracer* and *Dalvik instruction tracer* provide detailed accounts of the analysis sample's execution, while the *API tracer* provides a high level view of how the sample interacts with the rest of the system. The *taint tracker* implements dynamic taint analysis on native instructions but is capable of tracking taint through Java Objects with the help of the Dalvik view reconstruction. These tools were used to instrument and analyze two real-world malware samples: DroidKungFu and DroidDream.

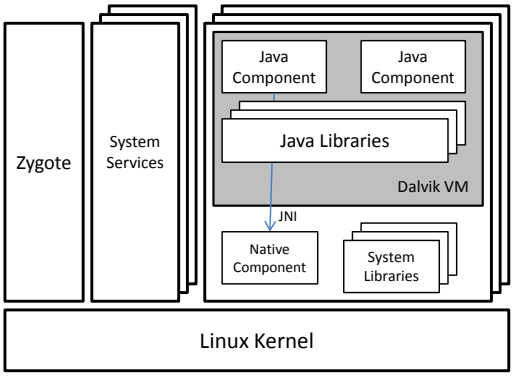

Figure 1: Overview of Android System

2 Background and Motivation

In this section, we give an overview of the Android system and existing Android malware analysis techniques to motivate our new analysis platform.

2.1 Android System Overview

Figure 1 illustrates the architecture of the Android system from the perspective of a system programmer. At the lowest level, the Android system uses a customized Linux kernel to manage various system resources and hardware devices. System services, native applications and Apps run as Linux processes. In particular, Zygote is the parent process for all Android Apps. Each App is assigned its own unique user ID (`uid`) at installation time and group IDs (`gids`) corresponding to requested permissions. These `uids` and `gids` are used to control access to system resources (i.e, network and file system) like on a normal Linux system.

All Apps can contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime. The Dalvik virtual machine (DVM), a shared library named *libdvm.so*, is then used to provide a Java-level abstraction for the App's Java components. At the same time, the Java Native Interface (JNI) is used to facilitate communications between the native and Java sides.

To create a Java component, an App developer first implements it in Java, compiles it into Java bytecode, and then converts it into Dalvik bytecode. The result is a Dalvik executable called a *dex* file. The developer can also compile native code into shared libraries, *.so* files, with JNI support. The dex file, the shared libraries and any other resources, including the *AndroidManifest.xml* file that describes the App, are packaged together into an *apk* file for distribution.

For instance, DroidKungFu is a malicious puzzle

game found in alternative marketplaces [25]. Its Java component exfiltrates sensitive information and awaits commands from the bot master. Its native component is used as a shell to execute those commands and it also includes three resource files that are encrypted exploits targeting known vulnerabilities, adb setuid exhaustion and udev [12], in certain versions of Android.

For security analysts, once a new Android malware instance has been identified, it is critical to quickly reveal its malicious functionality and understand its inner-workings. This often involves both static and dynamic analysis.

2.2 Android Malware Analysis

Like malware analysis on the desktop environment, Android malware analysis techniques can fall into two categories: static and dynamic. For static analysis, the sample's dex file can be analyzed by itself or it can be disassembled and further decompiled into Java using tools like *dex2jar* and *ded* [13]. Standard static program analysis techniques (such as control-flow analysis and data-flow analysis) can then be performed. As static analysis can give a complete picture, researchers have demonstrated this approach to be very effective in many cases [20].

However, static analysis is known to be vulnerable to code obfuscation techniques, which are commonplace for desktop malware and are expected for Android malware. In fact, the Android SDK includes a tool named Proguard [34] for obfuscating Apps. Android malware may also generate or decrypt native components or Dalvik bytecode at runtime. Indeed, DroidKungFu dynamically decrypts the exploit payloads and executes them to root the device. Moreover, researchers have demonstrated that bytecode randomization techniques can be used to completely hide the internal logic of a Dalvik bytecode program [14]. Static analysis also falls short for exploit diagnosis, because a vulnerable runtime execution environment is needed to observe and analyze an exploit attack and pinpoint the vulnerability.

Complementary to static analysis, dynamic analysis is immune to code obfuscation and is able to see the malicious behavior on an actual execution path. Its downside is lack of code coverage, although it can be ameliorated by exploiting multiple execution paths [6, 9, 31]. The Android SDK includes a set of tools, such as `adb` and `logcat`, to help developers debug their Apps. With JDWP (Java Debug Wire Protocol) support, the debugger can even exist outside of the device. However, just like how desktop malware detects and disables debuggers, malicious Android Apps can also detect the presence of these tools, and then either evade or disable the analysis. The fundamental reason is that the debugging components and malware reside in the same execution environment with the same privileges.

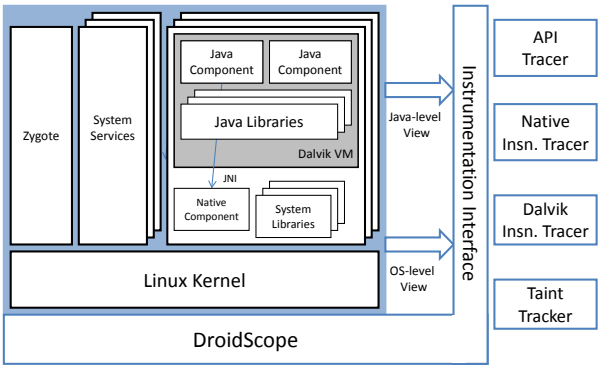

Figure 2: DroidScope Overview

Virtualization based analysis has proven effective against evasion, because all of the analysis components are out of the box and are more privileged than the runtime environment being analyzed, including the malware. Based on dynamic binary translation and hardware virtualization techniques, several analysis platforms [1, 15, 38] have been built for analyzing desktop malware. These platforms are able to bridge the semantic gap between the hardware-level view from the virtual machine monitor and the OS-level view within the virtual machine using virtual machine introspection techniques [16, 21, 24].

However, these tools cannot be immediately used for Android malware analysis. Android has two levels of semantic views, OS and Java, that need to be reconstructed versus the one for desktop malware. To enable virtualization-based analysis for Android malware, we need a unified analysis platform that reconstructs these two levels of views simultaneously and seamlessly binds these two views such that interactions between Java components and native components can be monitored and analyzed.

3 Architecture

DroidScope's architecture is depicted in Figure 2. The entire Android system (including the malware) runs on top of an emulator, and the analysis is completely performed from the outside. By integrating the changes into the emulator, the Android system remains unchanged and different virtual devices can be loaded. To ensure the best compatibility with virtual Android devices, we extended the QEMU [3] based Android emulator that ships with the Android SDK. This is done in three aspects: 1) we introspect the guest Android system and reconstruct OS-level and Java-level views simultaneously; 2) as a key binary analysis technique, we implement dynamic taint analysis; and 3) we provide an analysis interface to help analysts build custom analysis tools. Furthermore, we made similar changes to a different version of QEMU

to enable x86 support.

To demonstrate the capabilities of DroidScope, we have developed several analysis tools on it. The *API tracer* monitors the malware's activities at the API level to reason about how the malware interacts with the Android runtime environment. This tool monitors how the malware's Java components communicate with the Android Java framework, how the native components interact with the Linux system, and how Java components and native components communicate through the JNI interface.

The *native instruction tracer* and *Dalvik instruction tracer* look into how a malicious App behaves internally by recording detailed instruction traces. The Dalvik instruction tracer records Dalvik bytecode instructions for the malware's Java components and the native instruction tracer records machine-level instructions for the native components (if they exist).

The *taint tracker* observes how the malware obtains and leaks sensitive information (e.g., GPS location, IMEI and IMSI) by leveraging the taint analysis component in DroidScope. Dynamic taint analysis has been proposed as a key technique for analyzing desktop malware particularly with respect to information leakage behavior [18, 39]. It is worth noting that DroidScope performs dynamic taint analysis at the machine code level. With semantic knowledge at both OS and Java levels, DroidScope is able to detect information leakage in Java components, native components, or even collusive Java and native components.

We have implemented DroidScope to support both ARM and x86 Android systems. Due to the fact that the ARM architecture is most widely used for today's mobile platforms, we focus our discussion on ARM support, which is also more extensively tested.

4 Semantic View Reconstruction

We discuss our methodology for rebuilding the two levels of semantic views in this section. We first discuss how information about processes, threads, memory mappings and system calls are rebuilt at runtime. This constitutes the OS-level view. Then from the memory mapping, we locate the Dalvik Virtual Machine and further rebuild the Java or Dalvik-level view.

4.1 Reconstructing the OS-level View

The OS-level view is essential for analyzing native components. It also serves a basis for obtaining the Java-level view for analyzing Java components. The basic techniques for reconstructing the OS-level view have been well studied for the x86 architecture and are generally known as virtual machine introspection [16, 21, 24]. We employ similar techniques in DroidScope. We begin by first describing our changes to the Android emulator to enable basic instrumentation support.

Basic Instrumentation QEMU is an efficient CPU emulator that uses dynamic binary translation. The normal execution flow in QEMU is as follows: 1) a basic block of guest instructions is disassembled and translated into an intermediate representation called TCG (Tiny Code Generator); 2) the TCG code block is then compiled down to a block of host instructions and stored in a code cache; and 3) control jumps into the translated code block and guest execution begins. Subsequent execution of the same guest basic blocks will skip the translation phase and directly jump into the translated code block in the cache.

To perform analysis, we need to instrument the translated code blocks. More specifically, we insert extra TCG instructions during the code translation phase, such that this extra analysis code is executed in the execution phase. For example. in order to monitor context switches, we insert several TCG instructions to call a helper function whenever the translation table registers (system control co-processor c2_base0 and c2_base1 in QEMU) are written to.

With basic instrumentation support, we extract the following OS-level semantic knowledge: system calls, running processes, including threads, and the memory map.

System Calls A user-level process has to make system calls to access various system resources and thus obtaining its system call behavior is essential for understanding malicious Apps. On the ARM architecture, the service zero instruction *svc #0* (also known as *swi #0*) is used to make system calls with the system call number in register `R7`. This is similar to x86 where the *int 0x80* instruction is used to transition into privileged mode and the system call number is passed through the `eax` register.

To obtain the system call information, we instrument these special instructions, i.e. insert the additional TCG instructions, to call a callback function that retrieves additional information from memory. For important system calls (e.g. open, close, read, write, connect, etc.), the system call parameters and return values are retrieved as well. As a result, we are able to understand how a user-level process accesses the file system and the network, communicates with another process, and so on.

Processes and Threads From the operating system perspective, Android Apps are user-level processes. Therefore, it is important to know what processes are active and which one is currently running. In Linux kernel 2.6, the version used in Gingerbread (Android 2.3), the basic executable unit is the task which is represented by the `task_struct` structure. A list of active tasks is maintained in a `task_struct` list which is pointed to by `init_task`. To make this information readily available to analysis tools, DroidScope maintains a shadow task

list with select information about each task.

To distinguish between a thread and a process, we gather a task's process identifier `pid` as well as its thread group identifier `tgid`. The `pgd` (the page global directory that specifies the memory space of a process), `uid` (the unique user ID associated with each App), and the process' name are also maintained as part of the shadow task list. Additionally, our experience has shown that malware often escalates its privileges or spawns child process(es) to perform additional duties. Thus, our shadow task list also contains the task's credentials, i.e. `uid, gid, euid, egid` as well as the process' parent `pid`.

Special attention is paid to a task's name since the `comm` field in `task_struct` can only store up to 15 characters. This is often insufficient to store the App's full name, making it difficult to pinpoint a specific App. To address this issue, we also obtain the complete application name from the command line `cmdline`, which is pointed to by the `mm_struct` structure pointed to by `task_struct`. Note that the command line is located in user-space memory, which is not shared like kernel-space memory where all the other structures and fields reside. To retrieve it, we must walk the task's page table to translate the virtual address into a physical one and then read it based on the physical address.

According to the design of the Linux kernel, the `task_struct` for the current process can be easily located. The current `thread_info` structure is always located at the (*stack pointer & 0x1FFF*), and `thread_info` has a pointer pointing to the current `task_struct`. We iterate through all active tasks by following the doubly linked `task_struct` list. We also update our shadow list whenever the base information changes. We do this by monitoring four system calls *sys_fork*, *sys_execve*, *sys_clone* and *sys_prctl*, and updating the shadow task list when they return.

Memory Map The Dalvik Virtual Machine, libraries and dex files are all memory mapped and we rely on the knowledge of their memory addresses for introspection. Therefore, it is important to understand the memory map of an App. This is especially true for the latest version of Android, Ice Cream Sandwich, since address space layout randomization is enabled by default.

To obtain the memory map of a process, we iterate through the process' list of virtual memory areas by following the `mmap` pointer in the `mm_struct` pointed to by the `task_struct`. To ensure the freshness of the memory map information, we intercept the *sys_mmap2* system call and update the shadow memory map when it returns.

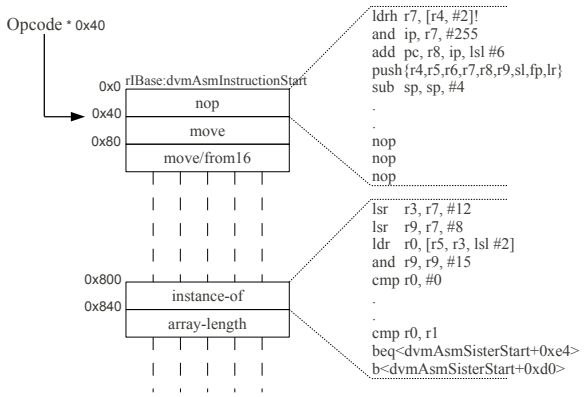

Figure 3: Dalvik Opcode Emulation Layout in *mterp*

4.2 Reconstructing the Dalvik View

With the OS-level view and knowledge of how the DVM operates internally, we are able to reconstruct the Java or Dalvik view, including Dalvik instructions, the current machine state, and Java objects. Some of the details are presented in this section.

Dalvik Instructions The DVM's main task is to execute Dalvik bytecode instructions by translating them into corresponding executable machine code. In Gingerbread and thereafter, it does so in two ways: interpretation and Just-In-Time compilation (JIT) [8].

The interpreter, named *mterp*, uses an offset-addressing method to map Dalvik opcodes to machine code blocks as shown in Figure 3. Each opcode has 64 bytes of memory to store the corresponding emulation code, and any emulation code that does not fit within the 64 bytes use an overflow area, `dvmAsmSisterStart`, (see `instance-of` in Figure 3). This design simplifies the emulation of Dalvik instructions. mterp simply calculates the offset, $opcode * 64$, and jumps to the corresponding emulation block.

This design also simplifies the reverse conversion from native to Dalvik instructions as well: when the program counter (`R15`) points to any of these code regions, we are sure that the DVM is interpreting a bytecode instruction. Furthermore, it is trivial to determine the opcode of the currently executing Dalvik instruction. In DroidScope we first identify the virtual address of rIBase, the beginning of the emulation code region, and then calculate the opcode using the formula $(R15 - rIBase)/64$. rIBase is dynamically calculated as the virtual address of *libdvm.so* (obtained from the shadow memory map in the OS-level view) plus the offset of `dvmAsmInstructionStart` (a debug symbol). If the debug symbol is not available, we can identify it using the signature for Dalvik opcode number `0` (`nop`).

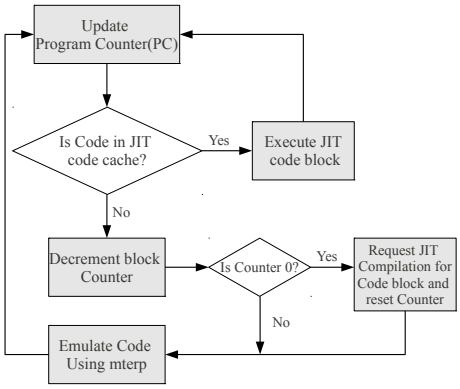

Figure 4: High Level Flowchart of *mterp* and JIT

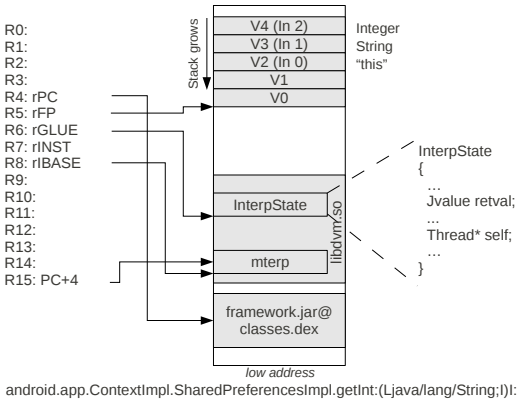

Figure 5: Dalvik Virtual Machine State

The Just-In-Time compiler was introduced to improve performance by compiling heavily used, or hot, Dalvik instruction traces (consisting of multiple code blocks) directly into native machine code. While each translation trace has a single entry point, there can be multiple exits known as *chaining cells*. These chaining cells either chain to other translation traces or to default entry points of the mterp interpreter. Overall, JIT provides an excellent performance boost for programs that contain many hot code regions, although it makes fine-grained instrumentation more difficult. This is because JIT performs optimization on one or more Dalvik code blocks and thus blurs the Dalvik instruction boundaries.

An easy solution would be to completely disable JIT at build time, but it could incur a heavy performance penalty and more importantly it require changes to the virtual device, which we want to avoid. Considering that we are often only interested in a particular section of Dalvik bytecode (such as the main program but not the rest of system libraries), we choose to *selectively* disable JIT at runtime. Analysis plugins can specify the code regions for which to disable JIT and as a result only the Dalvik blocks being analyzed incur the performance penalty. All other regions and Apps still benefit from JIT.

Figure 4 shows the general flow of the DVM. When a basic block of Dalvik bytecode needs to be emulated, the Dalvik program counter is updated to reflect the new block's address. That address is then checked against the translation cache to determine if a translated trace for the block already exists. If it does, the trace is executed. If it does not then the profiler will decrement a counter for that block. When this counter reaches 0, the block is considered hot and a JIT compilation requested. To prevent thrashing, the counter is reset to a higher value and emulation using mterp commences. As can be seen in the flow chart, as long as the requested code is not in the code cache, then mterp will be used to emulate the code.

The `dvmGetCodeAddr` function is used to determine whether a translated trace exists. It returns `NULL` if a trace does not exist and the address of the corresponding trace if it does. Thus, to selectively disable JIT, we instrument the DVM and set the return value of `dvmGetCodeAddr` to `NULL` for any translated trace we wish to disable. To show that our change to the virtual machine state does not have any ill side-effects, we make the following arguments. First, if the original return value was `NULL` then our change will not have any side effects. Second, if the return value was a valid address, then by setting it to `NULL`, the profile counter is decremented and if 0, i.e. the code region deemed hot again, another compilation request is issued for the block. In this case, the code will be recompiled taking up space in the code-cache. This can be prevented by not instrumenting the `dvmGetCodeAddr` call from the compiler.

In addition to preventing the translated trace from being executed, setting the value to `NULL` also prevents it from being chained to other traces. This is the desired behavior. For the special case where a translation trace has already been chained and thus `dvmGetCodeAddr` is not called, we flush the JIT cache whenever the disabled JIT'ed code regions change. This is done by marking the JIT cache as full during the next garbage collection event, which leads to a cache flush. While this is not a perfect solution, we have found it to be sufficient.

In all cases, the only side effect is wasted CPU cycles due to compilation; the execution logic is unaffected. Therefore, the side effects are deemed inconsequential.

DVM State Figure 5 illustrates how the DVM maintains the virtual machine state. When mterp is emulating Dalvik instructions, the ARM registers `R4` through `R8` store the current DVM execution context. More specifically, `R4` is the Dalvik program counter, pointing to the current Dalvik instruction. `R5` is the Dalvik stack frame pointer, pointing to the beginning of the current stack

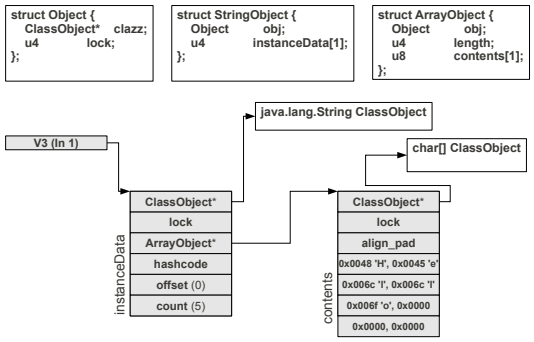

Figure 6: String Object Example

frame. `R6` points to the `InterpState` data structure, called `glue`. `R7` contains the first two bytes of the current Dalvik instruction, including the opcode. Finally `R8` stores the base address of the mterp emulation code for the current DVM instruction. In x86, `edx`, `esi`, `edi` and `ebx` are used to store the program counter, frame pointer, mterp base address and the first two bytes of the instruction respectively. The `glue` object can be found on the stack at a predefined offset.

Dalvik virtual registers are 32 bits and are stored in reverse order on the stack. They are referenced relative to the frame pointer `R5`. Hence, the virtual register `V0` is located at the top of the stack (pointed to by the ARM register `R5`,) and the virtual register `V1` sits on top of `V0` in memory, and so forth. All other Dalvik state information (such as return value and thread information) is obtained through `glue` pointed to by `R6`.

After understanding how DVM state is maintained, we are able to reconstruct the state from the native machine code execution. That is, by examining the ARM registers and relative data structures, we can get the current DVM program counter, frame pointer, all virtual registers, and so on.

Java Objects Java Objects are described using two data structures. Firstly, *ClassObject* describes a class type and contains important information about that class: the class name, where it is defined in a dex file, the size of the object, the methods, and the location of the member fields within the object instances. To standardize class representations, Dalvik creates a ClassObject for each defined class type and implicit class type, e.g. arrays. For example there is a ClassObject that describes a `char[]` which is used by `java.lang.String`. Moreover, if the App has a two dimensional array, e.g. `String[][]`, then Dalvik creates a ClassObject to describe the `String[]` and another to describe the array of the previously described `String[]` class.

Secondly, as an abstract type, *Object* describes a runtime object instance, i.e. member fields. Each Object has a pointer to the ClassObject that it is an instance of plus a *tail accumulator array* for storing all member fields. Dalvik defines three types of Objects, *DataObject*, *StringObject* and *ArrayObject* that are all pointed to by generic `Object*`s. The correct interpretation of any `Object*` fully depends on the ClassObject that it points to.

We use a simple String ("Hello") to illustrate the interpretation process. Figure 6 depicts the different data structures involved as well as the struct definitions on top. To access the String, we first follow the reference in the virtual register `V3`. Since Java references are simply `Object*`s, `V3` points to an Object. To determine the type of the object, we follow the first 4 bytes to the ClassObject structure. This ClassObject instance describes the `java.lang.String` class. Internally, Dalvik does not store the String data inside the StringObject and instead use a `char[]`. Consequently, `instanceData[0]` is used to store the reference to the corresponding `char[]` object and `instanceData[3]` is used to store the number of characters in the String, 5 in this case.

We then obtain the String's data by following `instanceData[0]` to the character array. Once again we must follow the `Object*` within the new object to correctly interpret it as an ArrayObject. Note that since ARM EABI requires all arrays to be aligned to its element size and `u8` is 8 bytes in length, we inserted an implicit 4 byte `align_pad` into the ArrayObject to ensure that the `contents` array is properly aligned. Given the length of the String from the StringObject and the corroborating length in the ArrayObject, the "Hello" String is found in the `contents` array encoded in UTF-16.

4.3 Symbol Information

Symbols (such as function name, class name, field name, etc.) provide valuable information for human analysts to understand program execution. Thus, DroidScope seeks to make the symbols readily available by maintaining a symbol database. For portability and ASLR support, we use one database of offsets to symbols per module. At runtime, finding a symbol by a virtual address requires first identifying the containing module using the shadow memory map, and then calculating the offset to search the database.

Native library symbols are retrieved statically through *objdump* and are usually limited to Android libraries since malware libraries are often stripped of all symbol information. On the other hand, Dalvik or Java symbols are retrieved dynamically and static symbol information through *dexdump* is used as a fallback. This has the advantage of ensuring the best symbol coverage for optimized dex files and even dynamically generated Dalvik bytecode.

	NativeAPI	LinuxAPI	DalvikAPI
Events	instruction begin/end	context switch	Dalvik instruction begin
	register read/write	system call	method begin
	memory read/write	task begin/end	
	block begin/end	task updated	
		memory map updated	
Query & Set	memory read/write	query symbol database	query symbol database
	memory r/w with pgd	get current context	interpret Java object
	register read/write	get task list	get/set DVM state
	taint set/check		taint set/check objects
			disable JIT

Table 1: Summary of DroidScope APIs

We rely on the data structures of DVM to retrieve symbols at runtime. For example, the *Method* structure contains two pointers of interest. `insns` points to the start of the method's bytecode, the symbol address, and `name` points to the name. Conveniently, the `glue` structure pointed to by `R6` has a field `method` that points to the Method structure for the currently executing method.

There are times when this procedure fails though, e.g. if the corresponding page of the dex file has not been loaded into memory yet. In these cases, we first try to look up the information in a local copy of the corresponding dex file, and if that fails as well, use the static symbol information from *dexdump*. DroidScope uses this same basic method of relying on the DVM's data structures to retrive class and field names as well.

5 Interface & Plugins

DroidScope exports an event based interface for instrumentation. We describe the general layout of the APIs, present an example of how tools are implemented, and finally describe available tools in this section.

5.1 APIs

DroidScope defines a set of APIs to facilitate custom analysis tool development. The APIs provide instrumentation on different levels: native, OS and Dalvik, to mirror the context levels of a real Android device. At each level, the analysis tool can register callbacks for different events, and also query or set various kinds of information and controls. Table 1 summarizes these APIs.

At the native level, one can register callbacks for instruction start and end, basic block start and end, memory read and write, and register read and write. One can also read and write memory and register content. As taint analysis is implemented at the machine code level, one can also set and check taint in memory and registers. Currently, the taint propagation engine only supports copy and arithmetic operations, control flow dependencies are not tracked.

At the OS level, one can register callbacks for context switch, system call, task start, update (such as process name), and end, and memory map update. One can also query symbols, obtain the task list, and get the current execution context (e.g., current process and thread). At the Dalvik level, one can instrument at the granularity of Dalvik instructions and methods. One can query the Dalvik symbols, parse and interpret Java objects, read and modify DVM state, and selectively disable JIT for certain memory regions. Through the Dalvik-view, one can also set and check taint in Java Objects as well.

5.2 Instrumentation Optimization

A general guideline for performance optimization in dynamic binary translation is to shift computation from the execution phase to the translation phase. For instance, if we need to instrument a function call at address x using basic blocks, then we should insert the instrumentation code for the block at x when it is being translated instead of instrumenting every basic block and look for x at execution time.

We follow this guideline in DroidScope. Consequently, our instrumentation logic becomes more complex. When registering for an event callback, one can specify a specific location (such as a function entry) or a memory range (to trace instructions or functions within a particular module). Therefore, our instrumentation logic supports single value comparisons and range checks for controlling when and where event callbacks are inserted during the translation phase.

The instrumentation logic is also dynamic, because we often want to register and unregister a callback at execution time. For example, when the virtual device starts, only the OS-view instrumentation is enabled so the Android system can start quickly as usual. When we start analyzing an App, instrumentation code is inserted to reconstruct the Dalvik view and to perform analysis as requested by the plugin. When instrumenting a function return, the return address will be captured from the link register `R14` at the function entry during execution, and a callback is registered at the return address. After the function has returned, this callback is removed. Then when the analysis has finished, other instrumentation code is removed as well. To maintain consistency, DroidScope invalidates the corresponding basic blocks in the translated code cache whenever necessary so that the new instrumentation logic can be enforced. Hence, the instrumentation logic in DroidScope is complex and dynamic. These details are hidden from the analysis plugins.

5.3 Sample Plugin

Figure 7 presents sample code for implementing a simple Dalvik instruction tracer. The *_init* function at L19 will be invoked once this plugin is loaded in DroidScope. In *_init*, it specifies which program to analyze by calling the

```
1. void opcode_callback(uint32_t opcode) {
2.   printf("[%x] %s\n", GET_RPC, opcodeToStr(opcode));
3. }
4.
5. void module_callback(int pid) {
6.   if (bInitialized || (getIBase(pid) == 0))
7.     return;
8.
9.   gva_t startAddr = 0, endAddr = 0xFFFFFFFF;
10.
11.  addDisableJITRange(pid, startAddr, endAddr);
12.  disableJITInit(getGetCodeAddrAddress(pid));
13.  addMterpOpcodesRange(pid, startAddr, endAddr);
14.  dalvikMterpInit(getIBase(pid));
15.  registerDalvikInsnBeginCb(&opcode_callback);
16.  bInitialized = 1;
17. }
18.
19. void _init() {
20.  setTargetByName("com.andhuhu.fengyinchuanshuo");
21.  registerTargetModulesUpdatedCb(&module_callback);
22. }
```

Figure 7: Sample code for Dalvik Instruction Tracer

setTargetByName function. It also registers a callback *module_callback* to be invoked when module information is updated. *module_callback* will check if the DVM is loaded and if so, disable JIT for the entire memory space (L9 and L11.) It also registers a callback, *opcode_callback*, for Dalvik instructions. When invoked, *opcode_callback* prints the opcode information.

This sample code will print all Dalvik instructions for the specified App, including the main program and all the libraries. If we are only interested in the execution of the main program, we can add a function call like *getModAddr("example@classes.dex", &startAddr, &endAddr)* at L10. This function locates the dex file in the shadow memory map and stores its start and end addresses in the appropriate variables. The rest of the code can be left untouched.

5.4 Analysis Plugins

To demonstrate the capability of DroidScope for analyzing Android malware, we have implemented four analysis plugins: API tracer, native instruction tracer, Dalvik instruction tracer, and taint tracker.

API tracer monitors how an App (including Java and native components) interacts with the rest of the system through system and library calls. We first log all of the App's system calls by registering for system call events. We then build a whitelist of the virtual device's built-in native and Java libraries. As modules are loaded into memory, any library not in the whitelist is marked for analysis. We instrument the *invoke** and *execute** Dalvik bytecodes to identify and log method invocations, including those of the sample. The log contains the currently executing Java thread, the calling address, the method being invoked as well as a dump of its input parameters. Since Java Strings are heavily used, we try to convert all Strings into native strings before logging them. We then instrument the *move-result** bytecode instructions to detect when system methods return and gather the return values. We do not instrument any of the other bytecodes to improve performance. To log library calls from the App's native components, we register for the block end event for blocks that are located in the App's native components. When the callback for the block end event is invoked, we check if the next block is within the Apps native components or not. If not, we log this event.

Native instruction tracer registers ARM or x86 instruction callbacks to gather information about each instruction including the raw instruction, its operands (register and memory) and their values.

Dalvik instruction tracer follows the basic logic of the above example and logs the decoded instruction to a file in the *dexdump* format. The operands, their values and all available symbol information, e.g. class, field and method names, are logged as well.

Taint tracker utilizes the dynamic taint analysis APIs to analyze information leakage in an Android App. It specifies sensitive information sources (such as IMEI, IMSI, and contact information) as tainted and keeps track of taint propagation at the machine code level until they reach sinks, e.g. *sys_write* and *sys_send*. With the OS and Dalvik views, it further creates a graphical representation to visualize how sensitive information has leaked out. To construct the graph, we first identify function and method boundaries. Whenever taint is propagated, we add a node to represent the currently executing function or method and nodes for the tainted memory locations. Since methods operate on Java Objects, we further try to identify the containing Object and create a node for it instead of the simple memory location. Currently, we only do this check against the method's input parameters and the current Object, e.g. "this". Further improvements are left as future work.

To identify method boundaries, we look for matching *invoke** or *execute** and *move-result** Dalvik instructions. We do not rely on the *return** instructions since they are executed in the invokee context, which might not be instrumented, e.g. inside an API. Since there are multiple ways for native code to call and return from functions plus malicious code is known to jump into the middle of functions, we do not rely on native instructions to determine function boundaries. Instead, we treat the nearest symbol that is less than or equal to the jump target in the symbol database as the function.

6 Evaluation

We evaluated DroidScope with respect to efficiency and capability. To evaluate efficiency, we used 7 benchmark Apps from the official Android Market: AnTuTu Benchmark (ABenchMark) by AnTuTu, CaffeineMark by Ravi Reddy, CF-Bench by Chainfire, Mobile processor benchmark (Multicore) by Andrei Karpushonak, Benchmark by Softweg, and Linpack by GreeneComputing. We then

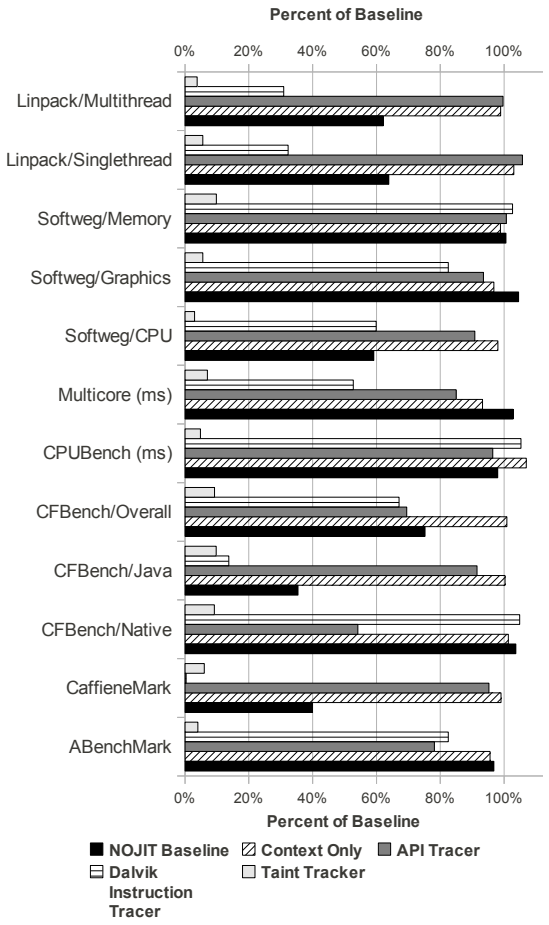

Figure 8: Benchmark Results

ran the benchmarks while using the different automatic analysis tools described above on the benchmarks themselves. The results are presented in Section 6.1. To evaluate capability, we analyzed two real world Android malware samples: DroidKungFu and DroidDream in detail, which will be presented in Sections 6.2 and 6.3. These samples were obtained from the Android Malware Genome project [40].

Experimental Setup All experiments were conducted on an Acer 4830TG with a Core i5 @ 2.40GHz and 3GB of RAM running Xubuntu 11.10. The Android guest is a Gingerbread build configured as "user-eng" for ARM with the Linux 2.6.29 kernel and uses the QEMU default memory size of 96 MB. No changes were made to the Android source.

6.1 Performance

To measure the performance impact of instrumentation, we took the analysis tools and targeted the benchmark Apps while the Apps performed their tests. This was repeated 5 times. As the baseline, we ran these benchmarks on the default Android emulator without any instrumentation. Since DroidScope selectively disables JIT on the Apps, we also obtained a `NOJIT` baseline with JIT completely disabled at build time. The performance results are summarized in the bar chart in Figure 8. Each tool is associated with a set of bars that shows its benchmark results (y-axis) relative to the baseline as a percentage. The ARM Instruction Tracer results are excluded as they are similar to the taint tracker results.

Please note that the benchmarks are not perfect representations of performance as evidenced by the $> 100\%$ results. For example, in CPUBenchmark the standard deviation, σ, for Baseline, Dalvik tracer and Context Only is only 1%. This means that the results are consistent for each plugin, but might not be across plugins. Furthermore, we removed the Softweg filesystem benchmarking results due to high variability, $\sigma > 27\%$.

We can see from Figure 8 that the overhead (`Context Only`) of reconstructing the OS-level view is very small, up to 7% degradation. The taint tracker has the worst performance as expected, because it registers for instruction level events. The taint tracker incurs 11x to 34x slowdown, which is comparable to other taint analysis tools [10, 39] on the x86 architecture. A special case is seen in the Dalvik instruction tracer result for CaffeineMark. This result is attributed to the fact that the tracer dynamically retrieves symbol information from guest memory for logging.

The benefits of dynamically disabling JIT is evident in some Java based benchmarks such as Linpack, CF-Bench/Java and CaffeineMark. For those benchmarks, the API tracer's performance is greater than that of the NOJIT Baseline, despite the fact that instrumentation is taking place. This difference is due to Java libraries, such as String methods, still benefiting from JIT in the API tracer.

6.2 Analysis of DroidKongFu

The DroidKungFu malware contains three components. First, the core logic is implemented in Java and is contained within the `com.google.ssearch` package. This is the main target of our investigation. Second are the exploit binaries which are encrypted in the apk, decrypted by the Java component and then subsequently executed. Third is a native library that is used as a shell. It contains JNI exported functions that can run shell commands and is the main interface for command and control. Unfortunately the command and control server was unavailable at the time of our test and thus we did not analyze this feature.

Discovering the Internal Logic We began our investigation by running the API tracer on the sample and analyzing the log. We first looked for system calls of interest and found a *sys_open* for a file named "gjsvro".

```
getPermission {
  if checkPermission() then doSearchReport(); return
  if !isVersion221() then
    if getPermission1() then return
  if exists("bin/su" or "xbin/su") then
    getPermission2(); return
  if !isVersion221() then getPermission3(); return
}
```

Figure 9: getPermission Pseudocode

There was also a subsequent *sys_write* to the file from a byte array. We later found that this array is actually part of a Java ArrayObject which was populated by the *Utils.decrypt* method, which is part of DroidKungFu. Since *decrypt* takes a byte array as the parameter, we were able to search backwards and identify that this particular array was read from an asset inside the App's package file called "gjsvro". It means that during execution, DroidKungFu decrypts an asset from its package and generates the "gjsvro" file. We then found that DroidKungFu called *Runtime.exec* with parameters "chmod 4755" and the name of the file, making the file executable and setting the setuid bit. After that, it called *Runtime.exec* again for "su" which led to a *sys_fork*. Furthermore, the file path for "gjsvro" was then written to a `ProcessImpl OutputStream`, followed immediately by "exit". Since this stream is piped to the child's *stdin*, we know that the intention of "su" was to open a shell which is then used to execute "gjsvro" followed by "exit" to close the shell. This did not work though since "su" did not execute successfully.

Next we used the Dalvik instruction tracer to obtain a Dalvik instruction trace. The trace showed that the *decrypt* and *Runtime.exec* methods were invoked from a method called *getPermission2*, which was called from *getPermission* following a comparison using the result of *isVersion221* and some file existence checks. To get a more complete picture of the *getPermission* method, we ran `dexdump` and built the overview pseudocode shown in Figure 9 . It is evident that to explore the *getPermission1* and *getPermission3*, we must instrument the sample and change the return values of the different method invocations.

With the Dalvik view support, we manipulated the return values of *isVersion221* and *exist* methods and were able to explore all three methods *getPermission1*, *getPermission2*, and *getPermission3*. They are essentially different ways to obtain the root privilege on different Android configurations. *getPermission1* and *getPermission2* only uses the "gjsvro" exploit. The main difference is that *getPermission1* uses *Runtime.exec* to execute the exploit while the other uses the "su" shell. On the other hand, *getPermission3* decrypts "ratc", "killall" (a wrapper for "ratc") and "gjsvro" and executes them using its own native library. As the API tracer monitors both

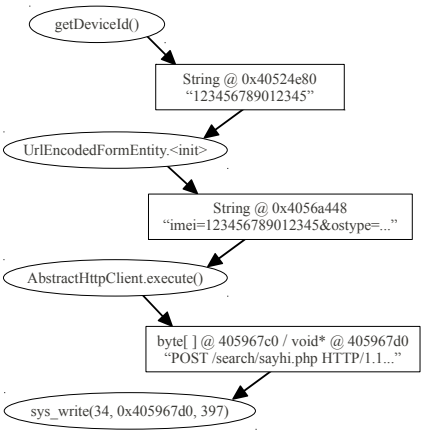

Figure 10: Taint Graph for DroidKungFu

Java and native components, our logs show that the library then calls *sys_vfork* and *sys_execve* to execute the commands. This indicates that *getPermission3* was trying to run both "udev" exploit and "rage against the cage" (ratc) exploits.

Analyzing Root Exploits Since Gingerbread has already been patched against these exploits, they never executed correctly. To further analyze these root exploits, we first needed to remove the corresponding patches from the virtual device build. Here we focus on "ratc," since "udev" is analyzed in the same manner. Due to space constraints we present the exploit diagnosis of "ratc" in Appendix A.

We first ran the API tracer on the ratc exploit, but did not observe any malicious behavior in the API log. We did see suspicious behavior in the process log provided as part of the OS-view reconstruction. Particularly, we observed that numerous ratc processes (descendants of the original ratc process) were spawned, the *adbd* process with uid 2000 ended, followed by more ratc processes and then by an adbd process with uid 0 or root. This signifies that the attack was successful. It is worth noting that the traditional adb based dynamic analysis would fail to observe the entire exploiting process because adbd is killed at the beginning.

Further analysis of the logs and descendent processes showed that there are in fact three types of ratc processes. The first is the original ratc process that simply iterates through the */proc* directory looking for the pid of the adbd process. Its child then forked itself until *sys_fork* returned -11 or EAGAIN. At this point it wrote some data to a pipe and resumed forking. In the grandchild process we see a call to *sys_kill* to kill the adbd process followed by attempts to locate the adbd process after it re-spawns.

Triggering Data leakage Reverting back to the default Gingerbread build, we sought to observe the information leakage behavior in *doSearchReport*. As depicted

in Figure 9, this involves instrumenting *checkPermission* during execution of *getPermission*. The Dalvik instruction trace shows that *doSearchReport* invokes *updateInfo*, which obtains sensitive information about the device including the device model, build version and IMEI amongst other things. We also observed outgoing HTTP requests, which failed because the server was down. We then redirected these HTTP requests to our own HTTP server by adding a new entry into */etc/hosts*. To further analyze this information leakage, we used the taint tracker and built a simplified taint propagation graph, which is shown in Figure 10. Objects, both Java and native, are represented by rectangular nodes while methods are represented by oval nodes. We see that *UrlEncodedFormEntity* (the constructor) propagated the original tainted IMEI number in the String @ 0x40524e80 to a second String that looks like an HTTP request. The taint then propagated to a byte array at 0x405967c0 by *AbstractHttpClient.execute*. We finally see the taint arriving at the sink at *sys_write*. Note that *sys_write* used a void* at 0x405967d0, which is the contents array of the byte array Object (see the StringObject example in Section 4.2). This is expected since JNI provides direct access to arrays to save on the cost of *memcpy*.

6.3 Analysis of DroidDream

Like analyzing DroidKungFu, we first used the API tracer to get a basic understanding of DroidDream, and then obtained instruction traces and analyzed information leakage.

From the log generated by the API tracer and the shadow task list, we found that there are two DroidDream processes. "com.droiddream.lovePositions," the main process, does not exhibit any malicious behavior except using *Runtime.exec* to execute "logcat -c" which clears Android's internal log. Again, this behavior indicates that traditional Android debugging tools fall short for malware analysis.

"com.droiddream.lovePositions:remote," the other process, is the malicious one. The logs show that DroidDream retrieves the IMSI number along with other sensitive information like IMEI, and encodes them into an XML String. Then we observed a failed attempt to open a network connection to `184.105.245.17:8080`. In order to observe this networking behavior, we instrument the return values of *sys_connect* and *sys_write* to make DroidDream believe these network operations are successful.

Using the taint tracker, we marked these information sources as tainted and obtained taint propagation graphs, which confirm that DroidDream did leak sensitive information from these sources to a remote HTTP server. The graph for leaking IMSI information is illustrated in Figure 11. We simplified the graph and annotated it to include *crypt* which is the DroidDream method used to xor-encrypt the byte array. The graph shows that *getSubscriberId* is used to obtain the IMSI from the system as a String @ 0x40522a10. The IMSI String, along with other information, is then encoded into an XML format using *format*. The resulting String is then converted into a byte[] @ 0x405232a8 for encryption by *crypt*. The encrypted version is used to create a *ByteArrayInputStream*. For brevity, we use a generic "API Native Memory" node to illustrate that the taint further propagates through memory until the eventual sink at *sys_write*.

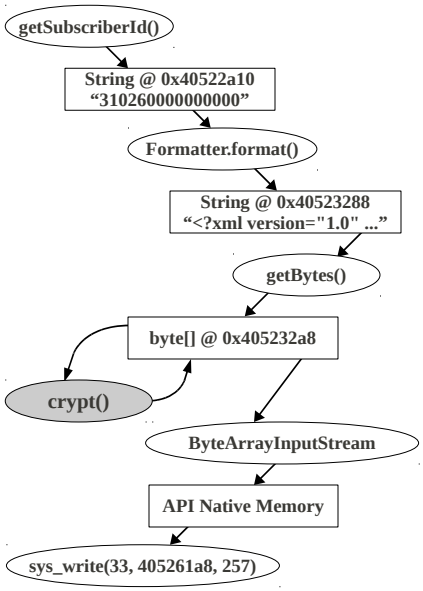

Figure 11: Taint Graph for DroidDream

We further investigated the *crypt* method by augmenting the Dalvik instruction tracer to track taint propagation and generate a taint-annotated Dalvik instruction trace. Not only do we see the byte array being xor-ed with a static field name "KEYVALUE," we also see that the encryption is being conducted on the byte[] in-place. A snippet of the trace log is depicted in Figure 12.

DroidDream also includes the udev and ratc exploits (unencrypted), plus the native library terminal like DroidKungFu. Since we have already analyzed them in DroidKungFu, we skipped the analysis on them in DroidDream.

7 Discussion

Limited Code Coverage Dynamic analysis is known to have limited code coverage, as it only explores a single execution path at a time. To increase code coverage, we may explore multiple execution paths as demonstrated in previous work [6, 9, 31]. In the experiments, we demonstrated that we can discover different execution paths by manipulating the return values of system calls, native

```
[43328f40] aget-byte v2(0x01), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 40523372(2401372)
  Adding M@410accec(42c5cec) len = 4
[43328f44] sget-object v3(0x0000005e), KEYVALUE// field@0003
[43328f48] aget-byte v3(0x88), v3(0x4051e288), v1(58)
[43328f4c] xor-int/2addr v2(62), v3(41)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@410accec(42c5cec) len = 4
[43328f4e] int-to-byte v2(0x17), v2(23)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@410accec(42c5cec) len = 4
[43328f50] aput-byte v2(0x17), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@40523372(2401372) len = 1
```

Figure 12: **Excerpt of Dalvik Instruction Trace for DroidDream.** A Dalvik instruction entry shows the location of the current instruction in square brackets, the decoded instruction plus the values of the virtual registers in parenthesis. A taint log entry is indented and shows tainted memory being read or written to. The memory's physical address is shown in parenthesis and the total bytes tainted is represented by "len."

APIs and even internal Dalvik methods of the App. This simple approach works fairly well in practice although a more systematic approach is desirable. One method is to perform symbolic execution to compute path constraints and then automatically explore other feasible paths. We have not yet implemented symbolic execution and leave it as future work. In particular, we seek to use tainting in conjunction with the Dalvik view to implement a symbolic execution engine at the Dalvik instruction level.

Detecting and Evading DroidScope In the desktop environment, malware becomes increasingly keen to the execution environment. Emulation-resistant malware detect if they are running within an emulated environment and evade analysis by staying dormant or simply crashing themselves. Researchers have studied this problem for desktop malware [2, 26, 36]. The same problem has not arisen for Android malware analysis. However, as DroidScope or similar analysis platforms become widely adopted to analyze Android malware, we anticipate similar evasion techniques will eventually appear. As malware may detect the emulated environment using emulation bugs in the emulator, some efforts have been made to detect bugs in the CPU emulators and thus can improve emulation accuracy [28, 29].

More troubling are the intrinsic differences between the emulated environment and mobile systems. Mobile devices contain numerous sensors, e.g. GPS, motion and audio, with performance profiles which might be difficult to emulate. While exploring multiple execution paths may be used to bypass these types of tests, they might still not be sufficient. For example we have observed that Android, as an interactive platform, can be sensitive to the performance overhead due to analysis. If the analysis takes too long, certain timeout events are triggered leading to different execution paths. The analyst must be aware of these new challenges. In summary, further investigation in this area is needed.

8 Related Work

Virtual Machine Introspection Virtual Machine Introspection is a family of techniques that rebuild a guest's context from the virtual machine monitor [21, 24]. This is achieved by understanding the important kernel data structures (such as the task list) and extracting important information from these data structures. For closed-source operating systems, it is difficult to have complete understanding of the kernel data structures. To solve this problem, Dolan-Gavitt et al. developed a technique that automatically generates introspection tools by first monitoring the execution of a similar tool within the guest system and then mimicking the same execution outside of the guest system [16]. With deep understanding of the Android kernel, DroidScope is able to intercept certain kernel functions and traverse proper kernel data structures to reconstruct the OS level view. In comparison, DroidScope takes it one step further to reconstruct the Dalvik/Java view, such that both Java and native components from an App can be analyzed simultaneously and seamlessly.

Dynamic Binary Instrumentation PIN [27], DynamoRIO [5], and Valgrind [32] are powerful dynamic instrumentation tools that analyze user-level programs. They are less ideal for malware analysis, because they share the same memory space with user-level malware and thus can be subverted. Bernat et al. used a formal model to identify observable differences due to instrumentation of *sensitive* instructions and created a *sensitivity-resistant* instrumentation tool called SR-Dyninst [4]. Like the other tools though, it cannot be used to analyze kernel-level malware.

Anubis [1], PinOS [7], TEMU [35], and Ether [15] are based on CPU emulators and hypervisors. They have the full system view of the guest system and thus are better suited for malware analysis. These systems only support the x86 architecture and Ether, in principle, cannot support ARM, because it relies on the hardware virtualization technology on x86. A new port must be developed for ARM virtualization [30]. While Atom based mobile platforms are available, ARM still dominates the Android market and thus ARM based analysis is important. To the best of our knowledge, DroidScope is the first fine-grained dynamic binary instrumentation framework that supports the ARM architecture and provides a comprehensive interface for Android malware analysis. We do not however support control flow tainting or different tainting profiles like Dytan [10]. Since Dytan is

based on PIN, it is theoretically feasible to port the tool to PIN for ARM [23], although it will still be limited to analyzing user-level malware.

Dalvik Analysis Tools Enck et al. used *ded* to convert Dalvik bytecode into Java bytecode and *soot* to further convert it into Java source code to identify data flow violations [20]. While powerful, the authors note that some violations could not be identified due to code recovery failures. DroidRanger is a static analysis tool that operates on Dalvik bytecode directly and was successful in identifying previously unknown malicious Apps in Android marketplaces [41]. TaintDroid and DroidBox are two examples of dynamic analysis tools for Android applications [17, 19]. TaintDroid is a specially crafted DVM that supports taint analysis of Dalvik instructions and across API calls. DroidBox is a project that uses TaintDroid to build an android application sandbox for analysis purposes. The biggest advantage of using TaintDroid is that it runs on actual devices. All of the hardware, sensors, vendor software and unpredictable intricacies that come with a real device are there. This can't be achieved in an emulated environment. The major negative of all these tools is that they are limited to analyzing the Java portion of Apps. Thus, if there is a native component, like DroidKungFu has, they will not be able to fully analyze it.

9 Conclusion

We presented DroidScope, a fine grained dynamic binary instrumentation tool for Android that rebuilds two levels of semantic information: operating system and Java. This information is provided to the user in a unified interface to enable dynamic instrumentation of both the Dalvik bytecode as well as native instructions. In this manner, the analyst is able to reveal the behavior of a malware sample's Java and native components as well as interactions between them and the rest of the system as evidenced by the successful analysis of DroidKungFu and DroidDream using DroidScope. These capabilities are provided to the analyst without changing the guest Android system and particularly with JIT intact. Our performance evaluation showed the benefits of dynamically disabling JIT for targeted analysis such as API tracing. The overall performance seems reasonable as well.

Acknowledgements

We thank the anonymous reviewers for their insightful comments towards improving this paper. This work is supported in part by the US National Science Foundation NSF under Grants #1018217 and #1054605. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the NSF or the Air Force Research Laboratory.

References

[1] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[2] BALZAROTTI, D., COVA, M., KARLBERGER, C., KRUEGEL, C., KIRDA, E., AND VIGNA, G. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, February 2010).

[3] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (April 2005).

[4] BERNAT, A. R., ROUNDY, K., AND MILLER, B. P. Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 89–99.

[5] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO'03)* (March 2003).

[6] BRUMLEY, D., HARTWIG, C., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SONG, D. BitScope: Automatically dissecting malicious binaries. Tech. Rep. CS-07-133, School of Computer Science, Carnegie Mellon University, Mar. 2007.

[7] BUNGALE, P. P., AND LUK, C.-K. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), VEE '07, pp. 137–147.

[8] CHENG, B., AND BUZBEE, B. A JIT compiler for android's dalvik VM. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html, 2010. Google I/O.

[9] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2011).

[10] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)* (2007), pp. 196–206.

[11] CRANDALL, J. R., AND CHONG, F. T. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)* (December 2004).

[12] Cve-2009-1185. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1185.

[13] ded: Decompiling Android Applications. http://siis.cse.psu.edu/ded/index.html.

[14] Dynamic, metamorphic (and opensource) virtual machines. http://archive.hack.lu/2010/Desnos_Dynamic_Metamorphic_Virtual_Machines-slides.pdf.

[15] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), pp. 51–62.

[16] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 297–312.

[17] Droidbox: Android application sandbox. http://code.google.com/p/droidbox/.

[18] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)* (June 2007).

[19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[20] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[21] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)* (February 2003).

[22] Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. http://gartner.com/it/page.jsp?id=1848514, 2011.

[23] HAZELWOOD, K., AND KLAUSER, A. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (New York, NY, USA, 2006), CASES '06, ACM, pp. 261–270.

[24] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)* (October 2007).

[25] Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[26] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec'09)* (November 2009).

[27] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference* (june 2005).

[28] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012).

[29] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing cpu emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)* (2009), pp. 261–272.

[30] MIJAR, R., AND NIGHTINGALE, A. Virtualization is coming to a platform near you. Tech. rep., ARM Limited, 2011.

[31] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland'07)* (May 2007).

[32] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI* (2007), pp. 89–100.

[33] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006* (April 2006).

[34] Proguard. http://proguard.sourceforge.net.

[35] TEMU: The BitBlaze dynamic analysis component. http://bitblaze.cs.berkeley.edu/temu.html.

[36] YAN, L.-K., JAYACHANDRA, M., ZHANG, M., AND YIN, H. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the Eighth Annual International Conference on Virtual Execution Environments (VEE'12)* (March 2012).

[37] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (February 2008).

[38] YIN, H., AND SONG, D. Temu: Binary code analysis via whole-system layered annotative execution. Tech. Rep. UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[39] YIN, H., SONG, D., MANUEL, E., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)* (October 2007).

[40] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)* (San Francisco, CA, USA, May 2012), IEEE.

[41] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium* (San Diego, CA, February 2012).

A Trace-Based Exploit Diagnosis of "ratc"

In this section, we provide an example of exploit diagnosis using DroidScope and the ARM instruction tracer on "ratc". These results corroborate with publicly available information on "ratc" and the setuid exhaustion vulnerability.

We know that *adbd* is supposed to downgrade its privileges by setting its uid to AID_SHELL (2000), and yet adbd retained its root privileges after the attack. Thus, in an effort to identify the root cause of the vulnerability, we used DroidScope to gather an ARM instruction trace that includes both user and kernel code.

A simplified and annotated log is shown in Figure 13. In the log, the instruction's address comes first followed by a colon, the decoded instruction and then the operands. We have also indented the instructions to illustrate the relative stack depth.

The log begins when *setgid* returns from the kernel space and returns back to *adb_main* at address 0x0000c3a4. Almost immediately, the log shows *setuid* being called. After transitioning into kernel mode, we see *sys_setuid* being called followed by a call to *set_user*. Later we see *set_user* returning an error code 0xfffffff5 which is (-11 in 2's complement or -EAGAIN).

Tracing backwards in the log reveals that this error code was the result of the *RLIMIT_NPROC* check in *set_user*. This reveals why *setuid* failed to downgrade

```
  ;;;setgid returns from kernel back to adbd
  0000813c: pop {r4, r7}
  00008140: movs r0, r0
  00008144: bxpl lr : Read Oper[0]. R14, Val = 0xc3a5
  ;; Return back to 0xc3a4 (caller) in Thumb mode

;;;adbd_main sets up for setuid
0000c3a4: movs r0, #250
0000c3a6: lsls r0, r0, #3 : Write Oper[0]. R0, Val = 0x7d0
  ;; 250 * 8 = 0x7d0 = 2000 = AID_SHELL

  ...

  ;;;Start of setuid section
  ;;;  213 is syscall number for sys_setuid
  00008be0: push {r4, r7} : Write Oper[0]. M@be910bb8, Val = 0x7d0
    ;; push AID_SHELL onto the stack
  00008be4: mov r7, #213
  00008be8: svc 0x00000000
    ;; Make sys call

    ;;; === TRANSITION TO KERNEL SPACE ===

    ;;;sys_setuid then calls set_user in kernel mode

      ;;;inside sys_setuid
      ;; Has rlimit been reached?
      c0048944: cmp r2, r3  : Read Oper[0]. R3, Val = 300 Read Oper[1]. R2, Val = 300

      ;;; RLIMIT(300) is reached and !init_user so return -11
      c0048960: mvn r0, #10 : Write Oper[0]. R0, Val = 0xfffffff5
          ;; the return value is now -11 or -EAGAIN
      c0048964: ldmib sp, {r4, r5, r6, fp, sp, pc}

      ;;;Return back to sys_setuid which returns back to userspace

    ;;; === RETURN TO USERSPACE ===

  ;;;setuid continues
  00008bec: pop {r4, r7}
  00008bf0: movs r0, r0 : Read Oper[0]. R0, Val = 0xfffffff5
      ;; -11 is still here

;;;Return back to adb_main at 0xc3ac (the return address) above
;;; Immediately starts other work, does not check return code
0000c3ac: ldr r7, [pc, #356] : Read Oper[0]. M@0000c514, Val = 0x19980330
  Write Oper[0]. R7, Val = 0x19980330
    ;; 0x19980330 is _LINUX_CAPABILITY_VERSION
```

Figure 13: Annotated adbd trace

adbd's privileges. Further analysis of the log shows that the return value from *setuid* was not used by adbd nor was a call to *getuid* seen. The same applies to *setgid*. This indicates that adbd failed to ensure that it is no longer running as root. Thus, our analysis shows that the vulnerability is due to two factors, RLIMIT_NPROC and failure to check the return code by adbd.