DROIT: Dynamic Alternation of Dual-Level Tainting for Malware Analysis*

CHIWEI WANG AND SHIUHPYNG WINSTON SHIEH
Department of Computer and Information Science
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: cwwangabc@gmail.com; ssp@cs.nctu.edu.tw

Taint analysis for Android malware has received much attention in recent research. Existing taint techniques operate either at Java object level or at deeper instruction level. Object-level tracking is suitable for malware written in Java byte-code, but not for native ones. Instruction-level tracking captures the finest data flow. However, it leads to obscure semantic reconstruction and low performance. In this paper, we present DROIT, a taint tracker which dynamically alternates between object-level and instruction-level tracking on demands. DROIT tracks data flow at Java object level in general. When its Dalvik VM exits the byte-code execution, DROIT automatically switches to instruction-level tracking, and vice versa. The trigger-based DROIT can alternate between the two levels in an efficient manner, and can provide dual-level whole image of the data flow, rather than fragments. Tracking at the dual levels also eases the semantic reconstruction significantly. The experiment with Android information-stealing trojans showed that DROIT can handle Java-based malware, those composed in native code, and those alternating between the two levels (e.g., DroidKungFu), respectively.

Keywords: mobile security, malware analysis, taint analysis, information flow tracking, binary translation, Android operating system, Dalvik virtual machine

1. INTRODUCTION

Due to the effectiveness of dynamic taint analysis, the technique is quickly applied to the analysis of Android applications. This technique generally tracks the data flow caused by the subject program. This analysis can be done at the Java object level [1, 2] or the native instruction level [3]. The two classes of implementations have their distinctive pros and cons. Tracking data flow only between the Java objects requires less effort, and can be performed at a fast speed. However, its limited scope is incapable of dealing with native-code malware like DroidKungFu. On the other hand, machine-level tracking captures the finest data flow, however it has two drawbacks. First, it demands instrumentation on each machine instruction. This leads to a high performance overhead. Secondly, the data flow at a deep instruction level generally loses the semantic meaning of the upper abstraction level. Copying a string and copying a data object will never be the same from a high-level abstract perspective, but they cannot be easily distinguished from at machine code level.

To cope with the weaknesses of both approaches while retaining their advantages,
we proposed DROIT, an emulation-based taint tracking system mixing the two modes. Running the Android environment inside an emulator, DROIT dynamically determines whether current execution context is within the Dalvik VM. If so, the object-level taint tracking is chosen in favor of higher performance and easier semantics reconstruction. Whenever the Dalvik VM sandbox exits and native code starts to take over, DROIT switches to instruction-level instrumentation to ensure completeness.

The first challenge encountered is that the system emulator must be able to distinguish Dalvik byte-code execution from ordinary native code. In addition, any data flowing between byte-code and native code must be tracked. Hooking at the JNI layer is not sufficient because Dalvik VM itself does not rely on JNI to exchange data between the native code and byte-code. Thereby, the monitoring should be done at all the internal functions for data exchange. This requires a deep insight into the implementation of Dalvik. Secondly, the object-level taint tracking manages its own taint tags in a way different from the instruction-level tracking. Also, the taint tags must be propagated across the tracking mode transition. The timing and the approach of this transition need further discussion. These issues are resolved in this paper.

In brief, the major features of DROIT are as follows:

- DROIT performs whole system taint tracking in a novel way by dynamically switching between object and instruction level. As a result, DROIT can monitor binary code execution and has a broader scope than just the Dalvik VM.
- Since most of the time the Android OS executes programs written in Dalvik byte-code, the system performance will benefit from the light-weight object tracking.
- A malware behavior profiling tool is developed upon DROIT. DROIT successfully detects the privacy theft behavior of real Android spyware, such as DroidDream, GingerMaster, and DroidKungFu no matter what these behaviors are done in Java code or native execution.

2. RELATED WORK

Taint tracking can be implemented at different levels. One of the earliest approaches was proposed to track dynamic information flow between objects (processes, files, etc.) at kernel level to detect illicit information leakage [5-9]. With the debut of the whole-system emulator, malware analysis can be performed at the instruction level on Windows and IA-32 architectures by adding hardware extension [10] or pure software emulation techniques [11, 12].

Recent work [13, 14] parse manifests and constructs data flows between applications to check whether security policies are violated. The Java byte-code may also be analyzed to generate a fine-grained data flow [13]. However, these techniques only examine the behaviors inside the Dalvik VM. A rough, preliminary concept [15] was proposed for Android system-level information flow tracking. Yet, design, implementation, and evaluation were not given. Paranoid [4] is a framework replaying system calls and other non-deterministic events recorded at the smartphone on the backend server to separate heavy-weight analysis. Their contribution focuses on the separation, rather than the security analysis itself.

Scott et al. [16], Brumley et al. [17], LIFT [18], and Minemu [19] developed analy-
sis platforms for binary programs on ARM architectures. All these work explored effective means to accelerate information flow tracking inside a process. However, it is a trend for advanced malware to circumvent by crossing the process boundary. Ginger-Master is an example trojan program capable of executing native code on the Android system. These behaviors cannot be captured without a system-level tracker.

Both TaintDroid [1] and AppInspector [2] track information flow at object level inside Android Dalvik VM. TaintDroid also monitors JNI invocation and the Binder so that a coarse-grained tracking is achieved for data flow outside the process. However, these schemes still can be circumvented by crafting malware purely in native code. In fact, such malicious programs already appeared in July, 2011 [20]. DroidScope [3] performs instruction-level taint tracking with the price of performance downgrade. In addition, the introspection heavily depends on the version of the Android Linux kernel and the Dalvik VM. Instead, DROIT only activates instruction-level tracking when a native library takes over. Directly implemented inside the Dalvik VM, the object-level tracking in DROIT is more resilient to software updates. The system also operates more efficiently without sacrificing too much comprehensiveness. PTT [21] and Demand emulation [22] accelerates taint tracking by entering virtualization when accessed data are guaranteed clean. Focusing on the dynamically switching between instruction-level and object level tracking, DROIT is trying to solve a totally different issue.

3. SYSTEM DESIGN

Before proceeding into the overview to the system design, we give a brief introduction to the operation of the Dalvik VM. An Android application is composed of mainly DEX files, containing the Dalvik byte-code sequences. Initially, the VM execute the application with an interpreter, which is referred to as mterp in Android. To accelerate, Android extends the interpreter with the trace-JIT technique. The flowchart of main loop of the execution engine is depicted in Fig. 1 (a). Byte-code traces are profiled along the interpretation, and hot traces will be compiled into native code on-the-fly. A byte-code trace with its execution counter reaching the threshold will be selected for JIT compilation. Depending on the threshold, the ratio of the code executed natively to those interpreted by mterp varies. Fig. 1 (b) shows the detailed flowchart of mterp. As illustrated, a big switch-case is used to invoke the proper handler according to the fetched byte-code.
As aforementioned, DROIT combines the object-level and the instruction-level taint tracking. To keep the article concise, we will refer to the object-level taint tracking mechanism in DROIT as OLTT and the instruction-level tracker as ILTT. As depicted in Fig. 2, the code emulated by QEMU \[23\] can be roughly divided into two classes: those responsible for Dalvik byte-code execution and the others. The former, which is decorated with horizontal stripes in Fig. 2, consists of the Dalvik op-code handlers of mterp and the native code blocks generated by the JIT compiler. In DROIT, they are equipped with the ability of in-system OLTT, as every other previous object-level tracker \[1, 2\]. For As shown, only one single taint tag is attached to each primitive type and every object in Dalvik VM.

The rest of the guest OS, which are illustrated with dotted areas in Fig. 2, are in the scope of the instruction-level tracker. The tracking is done at byte-granularity in a separated thread for efficiency. The emulator will be informed of the entrance and the exit of OLTT so that ILTT can be enabled or disabled correspondingly. Also, the taint tags are also mapped between the OLTT and the ILTT due to their different granularities.

**Security Assumption**

However, the adoption of OLTT actually lowers the security of the system. OLTT tracks taint at the object level and is implemented in the Dalvik VM. If vulnerability is found in the Dalvik op-code handlers or the JIT code cache, OLTT will be circumvented. Thereby, the following assumption is made for the attacker.

**Assumption 1:** Dalvik op-code handlers of Dalvik byte-code and the code in the JIT code cache are invulnerable.

In DROIT, OLTT is enabled only in the two parts mentioned in assumption 1. Namely, all other execution is monitored by ILLT. Note that Assumption 1 requires only the instruction causing the control transfer to be outside these regions, not the whole exploit vector. Thereby, an exploit composing the shellcode within the JIT code cache but attacking a buffer overflow in mterp is still detectable in DROIT because mterp is monitored by ILTT.
We argue that Assumption 1 holds due to following facts. Android SDK provides two implementation options for its op-code handlers. One is written in pure assembly language, and the other is implemented in portable C language. DROIT chooses the C version as they are all short (10–50 lines) and simple. Their invulnerability can be programmatically, or even manually, verified. As for those dynamically generated code in the JIT code cache, more secure translation can be used to ensure the randomness of the code layout. For example, random register allocation and instruction selection can greatly reduce the possibility of return-oriented or return-to-libc attacks.

3.1 OLTT for Tainted Code Execution

To ease the implementation effort, DROIT bases its object-level tracking on the well-known TaintDroid [1] system. TaintDroid sets checkpoints on JNI interfaces and the Binder, which is a unified inter-process message exchanger in Android, to track information flow between processes. However, TaintDroid does not fully satisfy the requirement of malware profiling. TaintDroid tracks data such as IMEI or SMS messages instead of programs. On the contrary, DROIT is designed to profile the program behavior, and the program itself is considered as the taint source. To suit our need, necessary modifications of TaintDroid are required.

The tainted code execution refers to a special situation that the program being executed itself is tainted. In this situation, the data generated by the tainted program should be carefully treated if these data can be executed as code. Unfortunately this is indeed the case. An Android application can define new classes at runtime by dynamically composing DEX byte-code and then loading them in through internal Dalvik function Dalvik_dalvik_system_DexFile_openDexFile_bytearray. To monitor all class loading operations, a hook is implemented at the class loader to check the taint tag of every class definitions. Note that DROIT does not rely on the taint tags of the YAFFS file system provided by TaintDroid. Instead, it uses ILTT to track the taint tag of each sector in the emulated SD card. Thereby, the check is implemented by examining the tag of all sectors occupied by the loaded DEX or JAR file.

To completely capture all data flows generated by the tainted coded, DROIT propagates the taint tags more aggressively than usual. In the mode of tainted code execution, every object and Dalvik VM registers modified are always tainted. Therefore, once a program in a tainted DEX file is executed, all its outputs are tainted.

To realize the above propagation, the Dalvik op-code handlers and the JIT compiler in Fig. 2 are modified. Every Dalvik op-code handler is augmented with a taint operation to propagate the taint tag of the running class file to all outputs. The appending is achieved with a bytewise-OR instruction. Noting that the taint status of the class file is fixed before execution, we modified the JIT compiler so that it injects additional propagation code only when the class file is tainted. Thereby, the performance of the JIT compiled code is not influenced at all when the class file is clean.

3.2 Instruction-Level Tracking

ILTT highly depends on the ARM binary code emulation. Executing emulation and ILTT in an interleaved design causes a severe performance downgrade. To cope with this
disadvantage, we took a different approach to implement ILTT. In our scheme, all the taint propagation procedures are executed by a different thread instead, and the main emulator will be only responsible for mandating the procedure to be invoked and for delivering necessary information such as memory addresses to the analysis thread. In this way, much fewer instructions will be injected, and hence performance can be improved.

Fig. 3 illustrates the ARM emulator and the taint tracker of DROIT. The left side of the figure shows the code blocks generated by the binary translator of the ARM emulator. For conciseness, the emulation code is only depicted in abstraction. The ARM binary instructions are continuously translated to these code blocks to execute. Note that the translation process is only invoked once for a newly discovered code block. The right side is the separated taint tracking thread, and a queue [25] implemented with shared memory plays as the communication channel in-between.

We modify the binary translator to generate two kinds of additional code to do taint tracking. One of the kinds is the code to actually propagate taint tags, which is stored in the code cache shown in the bottom-right in the figure. It is generated simultaneously as the emulation code blocks on the left side. As shown in Fig. 3, each ARM instruction has its proper taint propagation rules. For example, the instruction `STMFD SP!, {R4, LR}` copies the content in register R4 and LR to the memory location SP and SP+4, and hence it causes information flows $MEM \leftarrow LR$ and $MEM \leftarrow R4$. On the other hand, the instruction `SUB SP, SP, #0x10` computes out of the value of SP and store the outcome back to SP, and it thereby creates no new information flow. To acquire the accessed memory addresses, the other kind of code, which is depicted in bold texts on the left side of Fig. 3, is injected in original emulation code blocks. The codes deliver the memory address to the taint tracking thread through the queue.

3.3 OLTT/ILTT Switching

In DROIT. OLTT is implemented in the Dalvik VM, and hence it is naturally activated for all Dalvik byte-code execution and automatically disabled when Dalvik VM exits from the byte-code execution. However, ILTT is built in the ARM emulator, and it is always running even if the emulated ARM processor is actually executing the instructions for byte-code execution. Obviously, this does not meet our expectation since OLTT

![Fig. 3. The design of ILTT in DROIT.](image-url)
is already tracking the data flow between Dalvik objects.

To disable ILTT at the proper moment, the emulator must be aware of the execution of Dalvik byte-code. As aforementioned, the byte-code is completely executed either by the Dalvik op-code handlers or by the JIT code. In DROIT different ILTT-disabling strategies are taken for them.

**ILTT-disabling for JIT code**

As for the JIT code, we place hooks in the emulator to monitor the allocation of the JIT code cache. The allocation can be found in the internal function `dvmCompilerSetUpCodeCache()` of the Dalvik VM. Once the allocation succeeds, the virtual addresses of the allocated memory pages will be mapped back by the emulator to their physical memory addresses and stored in a hash map. The ILTT code generator simply ignores the ARM instructions residing in these physical pages. The termination of processes is also monitored. On the exit of a Dalvik VM process, the physical pages occupied by its JIT cache will be removed from the hash map.

**ILTT-disabling for Dalvik op-code handlers**

To disable ILTT while Dalvik op-code handlers are being emulated, the emulator must be able to distinguish them from ordinary code. This is achieved by statically identifying the offsets of these handlers relative to the beginning of `libdvm.so`. They are sequentially placed and form a continuous region. As long as the emulator starts to execute code in this region, the ILTT code injection will be disabled. According to our inspection, the region only takes 28 Kbyte in the whole `libdvm.so` (688 Kbytes).

### 3.3.1 Taint tag transferring in-between

When data are transferred between the native code and the byte-code execution, their taint tags must also be transferred between OLTT and ILTT because OLTT and ILTT manage their own taint tags in different ways. However, to control every data transfer is challenging. Merely hooking the JNI API is not complete because the Dalvik VM (native code) itself does not rely on them to access Java objects. Thereby, the monitoring should be done in a deeper level.

Fortunately, after studying the implementation of Dalvik VM in detail, these internal functions are all identified. We further categorize them into five classes, and APIs in a class share the same taint tag transfer methodology. An overview on the data flows between Dalvik VM and native code is illustrated in Fig. 4. There are two kinds of data in Dalvik: primitive types and objects. Primitive types, which consist of `byte`, `char`, `int`, `float`, etc., can only be individually declared as local variables, and hence they will be stored on stack frames. As shown, the OLTT implementation (TaintDroid) stores the taint tag of each primitive type immediately its data. On the other hand, objects can only be allocated in the heap, and they can be roughly divided into data objects and array objects. A data object is a set of possibly different primitive types, which are continuously allocated in the heap. Like local variables, the member variables of an object instance are also followed by their taint tags in the storage space. An array object, however, can only contain data of the same primitive type, and the data are sequentially placed. For efficiency, TaintDroid only maintain a single taint tag for the whole array.
Monitoring in-between Data Flow

Fig. 4 also points out another important fact that different access interfaces are provided for primitive types and objects due to their distinctiveness. The individual primitive types, which are allocated in the stack, can only be accessed by native code as pushed function arguments or return value. Namely, their data are delivered to or from native functions in the ordinary passed-by-value way. According to our study, the gate-keeping functions to realize the communication are `dvmPlatformInvoke()` and the `dvmCallMethod()` family. These functions are hooked in DROIT so that the data flows between the Dalvik stack and native code can be completely captured. As long as `dvmPlatformInvoke()` is invoked, the OLTT taint tags of the arguments in the Dalvik stack will be exported to the ILTT taint map and vice versa for the `dvmCallMethod()` family. Note that for ILTT the primitive type and their taint tags in the Dalvik VM are nothing different from ordinary data in guest memory. Therefore, when data are passed from Dalvik to native code, ILTT will taint both of them with the value of the OLTT taint tag.

For data objects, Dalvik VM provides a unified set of functions `dvmGet/SetField-XXX()` for native code to access member variables of an object. However, the field is nothing more than an offset, and these functions merely access the location at the given offset within the object. No boundary check is done.

As for array objects, the natives code can access the array elements by either invoking the array copying functions (e.g. `GetStringRegion()`, `Get/SetByteArrayRegion()`, etc.) or directly manipulating the memory after acquiring the buffer address (e.g. `pinPrimitiveArray()`, `NewDirectByteBuffer()`, `GetDirectBufferAddress()`, etc.).

In summary, the access provided by `dvmPlatformInvoke()`, `dvmCallMethod()`, and array copying functions do not allow native code to manipulate the heap space directly, and hence these “gates” can be more easily monitored by placing hooks on them. However, monitoring the data flow caused by direct buffer access and object field access is more complicated. Allowing native code to access the heap directly provides better performance, yet it makes the approach which we used to monitor accesses to stack variables invalid because programs may access the heap space at arbitrary instants and locations. Thereby, different approaches must be taken for them.
Object Field Access

As aforementioned, Dalvik VM provides a set of functions for native code to access object fields. All native code should always access object fields through these functions. Thereby, these functions would be the most reasonable locations to place monitoring hooks. However, the idea above can be circumvented. In Fig. 5, the implementation of the internal function `GetFloatField()`, which is a representative example for all field-accessing functions, is listed. As shown, the argument `fieldID` that the function takes is merely a pointer to the structure `InstField`. The offset value in `InstField` is then passed to `dvmGetFieldFloat()`, in which the target address is computed and accessed directly. Since no boundary check is performed, malicious code can access arbitrary memory locations at will by passing illegal `InstField` structures.

To prevent this attack, the following modifications and security checks are done in the source code of the Dalvik VM.

1. Define the macro `USE_INDIRECT_REF` provided by Android SDK. Enabling this feature will cause Dalvik to pass objects to native code using indirect references. These references are always recorded in the local or the global `IndirectRefTable`.
2. In the implementation of every `dvmGetFieldXXX()` functions, the passed argument `obj` is matched against all entry in the local and the global `IndirectRefTable` to verify if it is a valid pointer to an existing object.
3. Also in `dvmGetFieldXXX()` functions, the argument `offset` must be within the object size. The object size is acquired in `obj->clazz->objectSize`.
4. In addition, the offset must not refer to the taint tag. Recall that TaintDroid keeps taint tags and object field data interlaced. An incorrect offset may lead to arbitrary access to these tags. DROIT simply prohibits any accesses to them from native code.

Although above modification will definitely slow down the access, the slight overhead is affordable in malware analysis.

Direct Buffer Access

Generally speaking, the native code will invoke special functions such as `GetString-`
Chars() or GetByteArrayElements() to acquire the pointer to the buffer. Invoking these functions will pin the array object in the memory to prevent them being garbage collected. Then, native code will be able to directly read or write data from/into the buffer. The native code should also invoke the corresponding buffer-relinquishing functions to unpin the object after it finishes manipulating the buffer.

Direct buffer manipulation can occur at any locations. Meanwhile, since OLTT taint tags of data objects and array objects are stored in the heap space, arbitrary accesses to the heap space from native code imposes a more serious threat on the security policy of the system. In DROIT, the problem is solved by replacing the direct heap access with buffer-copy operation. Namely, on invocations of these functions a copy of the original buffer is created outside the Dalvik heap, and the address of this side buffer is returned instead. Meanwhile, an OLTT-to-ILTT taint tag synchronization is performed. Then, native code will operate on the copied buffer, and the data flow will be tracked by ILTT. When native code invoke buffer-relinquishing functions, the content of the side buffer is stored back to the original array object, and another ILTT-to-OLTT taint tag synchronization is done. In this way, only two taint tag synchronizations are needed. In addition, the functionality provided by the original buffer access functions such as GetStringChars() or GetByteArrayElements() are still supported, and hence benign programs can work normally. Even if a malicious program does not follow the relinquishing convention, the integrity of the Dalvik heap can still be kept.

3.3.2 Tag transferring policy

Here we give the formal description to the transferring policy between OLTT and ILTT. Please note all the data flows are forced to be passed in a passed-by-value (copy) way.

**Primitive Types on the Stack and the Heap**

Let’s denote a primitive type on the Dalvik stack or heap as $x$ and its OLTT tag as $O(x)$. Since $x$ is always passed by value through dvmPlatformInvoke() between Dalvik and native code, we also denote the physical memory region occupied by the copy of $x$ as $x.copy$. Given $r$, which is a set of physical memory locations, we use $I(r)$ to denote the collection of ILTT tags of all memory locations in $r$. Note that $r$ includes the locations occupied by the data and their OLTT tags. For data flowing out of $x$, we enforce the following transfer policy.

$$I(x.copy) := O(x)$$

On the other hand, new primitives types can be generated or modified in Dalvik only through dvmCallMethodXXX() with initial values provided in the argument. Let’s denote the physical location of the argument corresponding to the to-be-generated Dalvik primitive type $x$ as $x.src$. The following policy is enforced on such a data flow.

$$O(x) := I(x.src)$$
Strings and Array Objects

The modified internal functions still make copy for buffer access. However, different transfer policies are needed since TaintDroid only keeps a single OLTT tag for the whole array. Let’s denote a string or an array in the Dalvik VM as \( s \) and its OLTT tag as \( O(s) \) as well. We denote the physical memory region occupied by the created copy of \( s \) as \( s.copy \). Since \( s \) is a collection of elements, we denote its \( i \)th element as \( s.copy[i] \). For data flowing out of \( s \), we enforce the following transfer policy.

\[
I(s.copy[i]) := O(s) \text{ for all } 0 \leq i \leq n \tag{3}
\]

And for data flowing into \( s \), the following policy is enforced.

\[
O(s) := I(s.src[0]) \cup I(s.src[1]) \cup \ldots \cup I(s.src[n]) \tag{4}
\]

where \( n \) is the number of elements in \( x \).

Policies (1) and (2) are pretty straightforward. They ensure that in ILTT the memory location storing the copied data always have the same ILTT tag values as the one of the OLTT tag of the original variable. Since only one OLTT taint tag is attached a Dalvik array or string, Policy (3) forces ILTT to assign the shared taintness to all bytes in the duplicated buffer. Policy (4) computes the union of taintness of all elements in the source buffer, whose elements are about to be copied into the Dalvik array or string. Although false positives are inevitably inherited from the OLTT design, the policies do not generate any false negatives.

Note that our ILTT implementation adopts a parallelized design. Namely, the taint propagation of ILTT is actually running on a separate thread from the emulator. Thereby, synchronization must be performed on the two threads when Policies (2) and (4) are applied since the status of \( I(.) \) may not be immediately ready. No synchronization is needed for Policies (1) and (3) because OLTT, which is done in the emulation thread, is always ahead of ILTT.

Preventing Circumvention

Despite every internal data access function is monitored, malicious native code may simply disregard the suggested JNI functions such as \( GetByteField() \) or \( SetCharField() \) and access the object field in the Dalvik heap space directly. For the security reason, such usages should be always considered illegal. However, it is also infeasible to prohibit all native code from accessing the heap since the Dalvik VM internal may access the heap (e.g. Garbage collection). To resolve this, DROIT checks the taint tag of the memory location pointed by the program counter. Hence, ILTT is aware of tainted native code execution. Since all files of the subject program are labeled as the taint source, any native libraries or dynamically generated native code will be also tainted. Thereby, DROIT can prohibit accesses caused by the subject program without disrupting the normal operation of the system.

A return-to-libc attack or a return-oriented attack is possible. However, to apply these attacks the execution must be transferred by tainted data (e.g. over-written return address, function pointers, etc.). ILTT is inherently capable of giving alarms when tainted data hijack program execution [35].
4. EVALUATION

In this section, we evaluate effectiveness and performance of DROIT. The implementation is based on the QEMU coming in the Android SDK 2.2.1. All the experiments are performed on a PC with one unit of Intel i7-2600K 3.40 GHz Quad-Core Processor and 8GB DDR3 RAM. The emulated ARM architecture is allocated with 256 MB RAM and a 1GB SD card.

4.1 Malware Profiling

As aforementioned, DROIT views IMEI, IMSI, and ICCID as special taint sources to detect illicit privacy leakage onto networks through the emulated NIC. We studied 18 Android Trojan applications to verify effectiveness of DROIT. Among them, DorDrae.d, Plangton.a, Glodream.a, and Pjapps.a are in the top ten largest families in the Android Malware Genome Project [36]. GingerMaster is the first prototype of native Android malware. In Table 1 representative propagated payload sent by the analyzed malicious applications are shown.

<table>
<thead>
<tr>
<th>Family</th>
<th>(#samples)</th>
<th>Representative Tainted Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>DorDrae.d</td>
<td>12</td>
<td>TCP 54736 -&gt; 80, [4031 bytes of encrypted data, partially tainted by IMEI and IMSI]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCP 54736 -&gt; 80, [3912 bytes of encrypted data, partially tainted by IMEI and IMSI]</td>
</tr>
</tbody>
</table>
| Plangton.a | 2 | TCP 80 -> 49218, "deviceId": "260310IMEI00000", "...
| Glodream.a | 2 | TCP 80 -> 59219, "slotname=a14e08b42544bde&u_w=480&msid=com.GoldDream.pg&cap=m%2Casjs=afma-sdk-a-v4.1-0...
| Pjapps.a | 1 | TCP 9033 -> 49935, "nt=201110191810&sim=260310IMEI00000&tel=15555218135&msi=com.GoldDream.pg&cap=m%2Casjs=afma-sdk-a-v4.1-0...
| GingerMaster | 1 | TCP 5535 -> 49502, "...Post to: http://client.mustmobile.com/report/first_run.do 0x0d 0x0a I/GameService( 303): paramet er: [uid=260310IMEI00000, imei=260310IMEI00000, imsi=310260IMSI00000, iccid=8901410321118510720, sms=other...

Variants from family Plangton.a perform identity theft immediately directly. As shown in Table 1, a packet containing the IMEI is sent out. The trojan from family DorDrae.d send encrypted data to remote servers. In Table 1 we can observe that certain parts of the cipher text are polluted by IMEI and IMSI. The stealing behavior is verified by decompiling and analyzing the Trojan manually. It turns out DES encryption methods are used to encrypt the data stolen from the device. Glodream.a also sends out a packet polluted by IMEI periodically. However, some of its malicious behaviors are triggered by specific events or actions. For example, the core activities of Pjapps.a are registered as a service listening for the android.intent.action.BOOT_COMPLETED intent. To trigger these programs human assistance is needed. GingerMaster, appearing in August, 2011, exploits a vulnerability existing on all Android versions till v2.3 (v2.3 included) to acquire root privilege. The information stolen by GingerMaster is also identified.
Case Study: DroidKungFu

DroidKungFu is a representative advanced Android malware. Since its appearance in Jun, 2011, this trojan evolved and spawned massive amount of variants [36]. The latest variants are almost implemented in pure native code. The tainted file and data flows captured by DROID are shown in Tables 2 and 3. However, there are too many tainted OLTT-to-ILTT data flow recorded because each file opening or writing operation causes a corresponding native call. To be concise, only representative records are listed. Java application stopped after the attempt to execute the injected executable secbino. Note that in this phase no tainted network traffic is ever captured.

<table>
<thead>
<tr>
<th>Table 2. Files tainted by DroidKungFu.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/data/data/com.alan.translate/shared_prefs/sstimestamp.xml</td>
</tr>
<tr>
<td>/data/data/com.alan.translate/shared_prefs/permission.xml</td>
</tr>
<tr>
<td>/data/data/com.alan.translate/mycfg.ini</td>
</tr>
<tr>
<td>/data/data/com.alan.translate/secbino</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. OLTT to ILTT taint tag transfer caused by DroidKungFu.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoked function</td>
</tr>
<tr>
<td>dvmCreateCstrFromString</td>
</tr>
</tbody>
</table>

The injected file contains the privilege-escalation exploit and the main body of the trojan program. However, we failed to execute secbino directly because we cannot reproduce the same vulnerability on our emulated environment. To further expose the behaviors of the trojan, we simulate a successful exploit by executing secbino with the root permission manually.

<table>
<thead>
<tr>
<th>Table 4. Files tainted by secbino of DroidKungFu.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/system/bin/secbin</td>
</tr>
<tr>
<td>/system/etc/.dhcpcd</td>
</tr>
<tr>
<td>/system/etc/.rild_cfg</td>
</tr>
<tr>
<td>/data/data/com.alan.translate/WebView.db</td>
</tr>
</tbody>
</table>

As shown in Table 4, several files are copied from the application local directory to system directories. A new binary executable .dhcpcd is injected. Noticing the relation between the file copy operation and the last data flows recorded, we strongly suspect that the Java application is directing the spawned .dhcpcd through the local network socket, which can be observed in the first 3 OLTT-to-ILTT records in Table 5, and the preceding numbers seem to be command codes. The above guess is verified by reversing the executable .dhcpcd. Another interesting fact is that in our experiment no network message is ever observed in any OLTT-to-ILTT records, yet taint network traffics are captured in Table 6.

It shows that DroidKungFu extensively utilize native codes instead of Java programs to perform its activities. Without the ILTT extension, pure OLTT solution is insufficient for the analysis. The tainted OLTT-to-ILTT data flows also demonstrate the simplicity of semantic reconstruction in DROIT.
Table 5. OLTt to ILTT Taint Tag Transfer

<table>
<thead>
<tr>
<th>Invoked function</th>
<th>Buffer content</th>
</tr>
</thead>
<tbody>
<tr>
<td>dvmPlatformInvoke</td>
<td>Function: connectimpl 0x2b01</td>
</tr>
<tr>
<td>GetByteArrayRegion</td>
<td>0x7f 0x00 0x00 0x01</td>
</tr>
<tr>
<td>GetByteArrayElements</td>
<td>5 /system/bin/rm /system/bin/secbin</td>
</tr>
<tr>
<td>GetByteArrayElements</td>
<td>6 exit</td>
</tr>
<tr>
<td>GetByteArrayElements</td>
<td>2 /data/data/com.alan.translate/WebView.db /system/etc/.dhcpcd</td>
</tr>
<tr>
<td>GetByteArrayElements</td>
<td>2 /data/data/com.alan.translate/mycfg.ini /system/etc/.rild.cfg</td>
</tr>
<tr>
<td>GetByteArrayElements</td>
<td>4 /data/data/com.alan.translate/WebView.db</td>
</tr>
</tbody>
</table>

Table 6. Packets tainted by DroidKungFu

<table>
<thead>
<tr>
<th>Destination (DNS name)</th>
<th>Packet content</th>
</tr>
</thead>
<tbody>
<tr>
<td>search.zs169.com</td>
<td>GET /search/isavaible2.php?imei=260310IMEI00 000&amp;ch=devluo&amp;ver=10 HTTP/1.0 User-Agent: ad lib/3 (Date: 1998/09/23 06:19:15 $)</td>
</tr>
<tr>
<td>search.gongfu-android.com</td>
<td>GET /search/isavaible2.php?imei=260310IMEI00 000&amp;ch=devluo&amp;ver=10 HTTP/1.0 User-Agent: ad lib/3 (Date: 1998/09/23 06:19:15 $)</td>
</tr>
<tr>
<td>search.zi18.com</td>
<td>GET /search/isavaible2.php?imei=260310IMEI00 000&amp;ch=devluo&amp;ver=10 HTTP/1.0 User-Agent: ad lib/3 (Date: 1998/09/23 06:19:15 $)</td>
</tr>
</tbody>
</table>

4.2 Performance

To measure the performance, we chose the open source Android benchmark project 0xbench, which includes a series of performance tests: Fast Fourier Transform, Jacobi successive over-relaxation, Monte Carlo integration, Sparse Matrix Multiplication, LU Matrix Factoring, and Dalvik Garbage Collection. The scores of the first five tests are given in approximate Mflops (Millions of floating point operations per second). The performance of the garbage collection is measured by the elapsed time in milliseconds.

(a) Pure emulator (neither ILTT nor OLTt presents)
(b) Tracked only by OLTt (TaintDroid)
(c) Fully and only tracked by ILTT
(d) DROIT (OLTt/ILTT switching)

Configuration (a) sets the baseline of the comparison. The original Android system is running on an unmodified QEMU, and we expect that it would run at the fastest speed. The pure OLTt performance is measured in Configuration (b), which executes the original TaintDroid project. Configuration (c) measures the performance of our ILTT implementation. Namely, ILTT is turned on for JIT and op-code handlers as well. Recall that our ILTT implementation adopts a parallelized design. Thereby, it will utilize an additional processor core. Lastly, we measure the performance of the dynamically ILTT/ OLTt design in configuration (d).

In the upper part of Table 7 is given the benchmark results of the 0xbench. Each test is repeated 20 times to acquire the average value $\mu$ and the standard variation $\sigma$. As expected, in all tests the unmodified emulator acquired the best scores, and the pure ILTT works at the worst performance. In addition, the following phenomena should be noticed.
Table 7. Performance evaluation with 0xbench under different configurations.

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Jacobi SOR</th>
<th>Monte Carlo</th>
<th>Sparse Matrix Mul.</th>
<th>LU Matrix Factoring</th>
<th>Dalvik GC (unit: ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>μ</td>
<td>σ</td>
<td>μ</td>
<td>σ</td>
<td>μ</td>
<td>σ</td>
</tr>
<tr>
<td>(a)</td>
<td>1.93</td>
<td>0.02</td>
<td>5.37</td>
<td>0.25</td>
<td>0.92</td>
<td>0.06</td>
</tr>
<tr>
<td>(b)</td>
<td>0.61</td>
<td>0.03</td>
<td>1.48</td>
<td>0.17</td>
<td>0.25</td>
<td>0.01</td>
</tr>
<tr>
<td>(c)</td>
<td>0.10</td>
<td>0.01</td>
<td>0.25</td>
<td>0.09</td>
<td>0.05</td>
<td>0.01</td>
</tr>
<tr>
<td>(d)</td>
<td>0.33</td>
<td>0.02</td>
<td>0.92</td>
<td>0.13</td>
<td>0.12</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Number of code blocks executed in each test. (unit: million)

| Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native | Dalvik Native |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 144.7         | 83.3          | 284.7         | 155.6         | 193.5         | 177.8         | 83.6          | 48.1          | 184.8         | 124.9         | 419.7         | 704.6         |

1. DROIT outperforms pure ILTT (c) significantly in all tests (2.33X~4.01X) except the garbage collection test (1.24X).
2. Even if only OLTT is enabled (b), the performance is still far behind the unmodified Android system (a).
3. Regarding the five tests, by computing the ratio of the average score acquired in configuration (d) to the one in configuration (b), it is estimated that DROIT maintains 49%~64% performance of the pure OLTT design. However, the ratio drops to 42% when garbage collection is performed.

In fact, all the above observations are expected since DROIT only switches to ILTT when the emulator executes codes not belonging to the Dalvik op-code handlers. The garbage collection causes the performance drop because the GC is not performed by Dalvik op-codes but the Dalvik VM itself, which is being tracked by ILTT.

However, to verify our guess we performed another experiment to accumulate the number of executed code blocks inside the Dalvik op-code handlers and the rest of the system respectively. The result is given in the lower part of Table 7. Note that the accumulation is acquired in a separate experiment, and hence the previous benchmark results are not affected. As indicated, the five tests implemented in pure Java language lead to more execution in Dalvik op-code handlers. On the contrary, the garbage collection executes more often in native codes.

5. LIMITATIONS

The OLTT of DROIT still suffers from false positives due to the tracking is done on the whole object. This is a limitation inherited from all object-level taint trackers. Also, DROIT does not automatically resolve triggering constraints. To satisfy a constraint-based trigger such as a time bomb or an emulation check, is however a different issue and is sometimes un-decidable as halting problems. To cope with these problems, results of research work for software testing [26-28] can be considered.

Implicit flows are a challenging issue in information flow tracking. This issue has been discussed in section 3.2. Since in DROIT the data generated by tainted code execution are always tainted, it is guaranteed that the subject program (either the DEX program or the native library) cannot directly utilize any implicit flows to evade our profiling. However, the detection of tainted code execution does not solve the program completely. There are other ways to circumvent the data flow tracking mechanism. In fact,
Cavallaro et al. [29] and Slowinska et al. [30, 32] had the insightful discussion on the limitation of information flow tracking. They point out the fundamental infeasibility of implicit flow tracking. However, Dalton et al. [31] defended the practicality of pointer tainting. As for control flows, McCamant et al. [33] modeled binary programs as graphs and measured their maximum information flow capacities. Newsome et al. [34] proposed a method to measure quantitative influence of a program was proposed. Their work gave in-depth discussion about these issues.

6. CONCLUSIONS

In this paper, we present DROIT, which dynamically switches between ILTT and OLTT for advanced Android malware analysis. The on-demand switching makes DROIT works as comprehensive as the conventional instruction-level tracking. Based on DROIT, a malware behavior profiling platform is constructed. In our experiment, DROIT successfully records the modification and the network traffic caused by the subject program. The case study on DroidKungFu further demonstrates the power of DROIT by capturing the tainted OLTT-to-ILTT data flows. The performance evaluation shows that the alternating design can benefit a lot from the light-weight OLTT. These evidences show that the proposed design is efficient and effective for Android malware analysis. With the ability to simultaneously trace Java-based malware, native-coded ones, and mixed hybrids, DROIT can assist analysts in antivirus industry in understanding behaviors of unknown advanced Android malware.

REFERENCES


Chiwei Wang (王維偉) is currently a Ph.D. candidate in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. He has been very active in the malicious software analysis community, and has received many awards. Recently, he led his team and achieved 1st place in the War-game Contest held by Hacks in Taiwan Conference 2010 and 2011. He also received the 1st place in the Microsoft Cross-Strait Innovation.
tion Contest in 2007. His research interests include network security, software security, and operating systems.

Shiuhpyng Winston Shieh (謝續平) received the M.S. and Ph.D. degrees in Electrical and Computer Engineering from the University of Maryland, College Park, respectively. He is a Professor of the Department of Computer Science, National Chiao Tung University (NCTU), and the Director of Taiwan Information Security Center at NCTU, and has been the Editor-in-Chief of IEEE Reliability Society Newsletter, and Associate Editor of IEEE Transactions on Reliability, IEEE Transactions on Dependable and Secure Computing, ACM Transactions on Information and System Security, Journal of Computer Security, Journal of Information Science and Engineering, and Journal of Computers, respectively. He was the program chair of IEEE Conference on Software Security and Reliability, steering committee chair and program chair of ACM Symposium on Information, Computer and Communications Security. He received the Distinguished Information Technology Award (presented by Vice President of Taiwan), and ACM Distinguished Scientist Award among the 41 recipients worldwide that year. His research interest includes network and system security.