2014 IEEE International Conference on Computer and Information Technology

DIAS: Automated Online Analysis for Android Applications

Juanru Li
Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China

Yuanyuan Zhang, Wenbo Yang
Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China

Junliang Shu, Dawu Gu
Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China

***Abstract*—Online program analysis aims to analyze a program as it executes. Traditional online program analysis is generally interactive and not automated. An automated online program analysis requires fine-grained yet flexible analyzing infrastructure to support. Android system, although providing many high-level debugging interfaces, lacks such functionality and is not convenient for developing automated analysis tools.**

In this paper we present DIAS, a flexible and extensible framework for automated online analysis of Android applications. DIAS contains an introspection monitor and many analysis modules. Based on the introspection monitor and the analysis modules, DIAS provides a set of analysis interfaces to help access various types of runtime information and performs automated online analysis. Automated program analysis is then accomplished using these interfaces.

DIAS can monitor applications' execution and understand their behavior, which is useful for profiling, debugging and security analysis. Moreover, DIAS's analysis interface eases the task of developing new online analysis functions.

I. Introduction

Android application nowadays are automated analyzed by many tools with in-depth static analysis techniques. However, online analysis(a.k.a real-time dynamic program analysis) technique of Android applications is less automated. Developing automated online program analysis tool is still a tough yet attractive task. Unlike binary code instrumentation and debugging techniques on commodity computer platform[18][10][14] which are well supported by fundamental infrastructure, Android platform's analyzing infrastructure such as ADB[1] and DDMS[7] currently lacks such automated analysis functionality for assessing and monitoring applications in a comprehensive way. A supportive framework is necessary to meet the requirements of automated online program analysis. An analysis framework boosts analyst's work efficiency by providing abundant runtime information in a concise style. The framework encapsulates most low-level details so that even the OS is upgraded frequently, the existing analysis tools are still available. And analysis based on this specific framework is more robust than analysis that directly operates on system components.

Designing an online analysis framework for Android application faces many challenges. Two essential factors of automated online program analysis are real-time monitoring and programmable analysis. Real-time monitoring requires to monitor every execution details at the same time the analyzed process is executing. And programmable analysis requires code/script to operate the analyzed object rather than human based interactive work. In detail, the proposed framework should contain following features: *(i)* To achieve the goal of monitoring details of execution, the framework should provide fine grained instrumentation on application for comprehensive analysis. Program analysis is often asked to find the root cause of a problem inside a complex execution and numerous instructions are to be analyzed. If the detail of the execution cannot be well monitored, analyst may fail to solve the problem. *(ii)* With the framework new analysis modules need not to be built from the ground-up. Effective developing styles such as using scripts and configuration file are expected to be supported. *(iii)* Due to the weaker processing capability of the mobile device, it is expected that the framework incurs reasonable overhead.

There are commonly two approaches for Android application's online program analysis. The first approach is using bytecode rewriting to monitor sensitive API calls and deploying policy enforcement. The advantage of this approach is that it does not modify original Android system and the overhead is reasonable. The disadvantage is that it is less fine-grained(for bytecode level monitoring) and it breaks the integrity of the application. The second approach is inserting analysis component to Android's Dalvik VM to monitoring the interpretation. This approach can monitor details of every bytecode execution and need no modification of the analyzed application. Although VM monitoring requires to modify the original system, it contains a number of attractive advantages. Monitoring at the VM layer brings minimal modification, and makes the architecture of framework compact. Monitoring at the VM layer provides better transparency for online analysis because analyzed app cannot be aware of the low-level monitoring, while bytecode rewriting approach is easier to bring heisenbugs[16] due to the modification of the code and may perturb application's behavior.

In this paper we present **DIAS**, an online program analysis framework for Android applications which helps analyst fulfil automated analysis. In general, DIAS contains an introspection monitor and many analysis modules. Inside the Dalvik VM of Android runtime, DIAS uses an introspection monitor to

978-1-4799-6239-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CIT.2014.82

collect detail runtime information of application's bytecode interpretation process at the Dalvik VM layer. The monitoring code is attached to every key processing point of Dalvik VM to inspect the execution. Outside the Dalvik VM, DIAS introduces analysis modules at the system service layer of Android to support automated analysis deal with the raw data collected by the introspection monitor, fulfilling complex task such as encapsulating the raw data into high-level form or analyzing the status of execution. Analysis modules can also export interfaces for other modules to use.

DIAS provides systematic methodology to achieve automated online program analysis of Android applications. DIAS supports automated analysis in two aspects: First, it provides different interfaces to help inspect runtime information or manipulate the execution process. Interfaces with simple function are provided by introspection monitor directly. Interfaces with complex function are exported by modules to keep introspection monitor compact. Second, the analysis module enables analyst to interact with interfaces automatically. The philosophy of DIAS encourages analyst to develop module to extend the analysis functions based on existing interface. In this manner, DIAS eases the task of developing new analysis tools. Analysts need not to constantly re-implement the low-level analysis functions and may efficiently construct complex analysis tools to accomplish tasks such as application behavior understanding and security checking.

DIAS can be viewed as an extension of Android's logging and debugging service. Rather than simply implementing online analysis, the main contribution of DIAS are:

- *Dalvik VM introspection*. One of the most important features of DIAS framework is that it analyze application through inspecting the Dalvik VM. Instead of inspecting system events only, DIAS could monitor details of every instruction's execution. This gives analyst powerful observation capability. Based on the introspection monitor, analysis modules can be used as information wrapper to assemble low-level data into high-level semantic-rich events, and perform advanced analysis functions.
- *Extensible Analysis Interface.* Interfaces of DIAS defines the function of the framework. An analysis task need not develop analyzing code from the ground-up and could take full advantage of the existing interfaces. Because DIAS provides different interface for dealing with different types of runtime state. Analyst could simply decide which part of execution should be analyzed by choosing proper interface. The combination of different interfaces generates various functions for debugging, profiling or security monitoring. Except for the pre-built interfaces, new analysis module with complex function can also be developed to extend the framework's interface.
- *Build-in Analysis.* Original Android debugging and logging infrastructure only supports remote interaction, which means the recorded information is observed and dealt outside the system. DIAS's analysis module executes within the analyzed applications in the same system. This build-in analysis inspects the execution with better efficiency and avoids interactive overhead.

II. DIAS Framework

A. Overview

Figure 1 shows the overview of DIAS framework. Inside the Dalvik VM is the introspection monitor of DIAS which monitors the execution of any analyzed application. The introspection monitor inserts instrumentation interfaces to wherever the bytecode is interpreted, monitors the execution process of the Dalvik VM. The VM-based monitoring guarantees transparency to the analyzed applications when they are executed with DIAS's introspection. Connected with the introspection monitor, analysis modules of DIAS fulfil certain analysis tasks using the information from interfaces exported by the monitor. The framework is designed for general purpose analysis and can meet the requirements of almost all sorts of automated online analysis on Android applications.

DIAS framework is compact for it uses one introspection core to monitor all kinds of information. A compact architecture makes DIAS independent of the change of system environment and persists in high consistency. Thus DIAS can support various kinds of Android devices well and is less affected by the upgrading of the operating system. And monitoring the Dalvik VM is efficient for collecting data with minimum modification of the system.

The hierarchy of DIAS(introspection monitor + analysis modules) isolates the analysis work and the execution of Dalvik VM. This is important for keeping VM work stable and proper. Moreover, analysis module can be extended by third party developers. It is much better for a framework to dynamically load new modules than integrate all sorts of functions in one module.

B. Dalvik Introspection

DIAS uses an introspection monitor to observe the Dalvik VM's state and to record information during the execution. An online analysis on the Dalvik VM can effectively monitor the execution of an application for two reasons: First, Dalvik VM is the core part of the Android OS which is responsible for interpreting the bytecode for the whole application layer of the Android OS stack. Thus the interpreter of the Dalvik VM is the essential place to monitor the execution of any application. Second, unlike commodity VM which often contains a full OS and one or more applications, in Android every application has its own Dalvik VM instance in its process space. The introspection, although from low level, has a good viewpoint focusing on one certain application's execution.

The introspection monitor consists of many "probes". The probe is actually instrumentation code inserting to the Dalvik VM to inspect it. These instrumentation code are then executed wherever the analyzed application is executed. The Dalvik VM includes two interpreters, a *portable* one and a *fast* one[11]. The portable interpreter is largely contained within a single C function, and can be compiled on any system that supports GCC. The fast interpreter uses hand-coded assembly fragments to boost the interpretation on specific hardware platform.

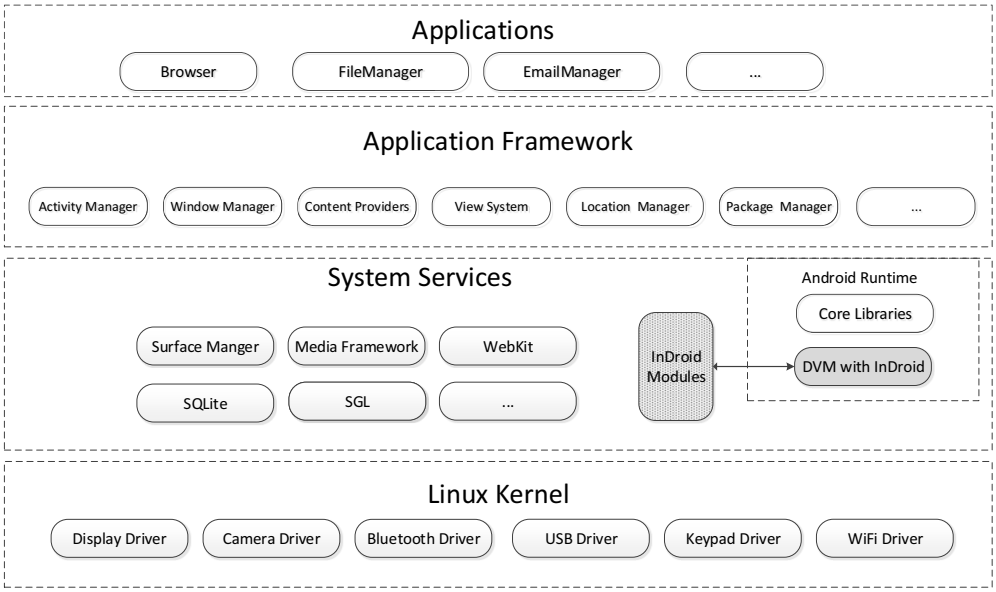

Fig. 1. DIAS Framework

DIAS's introspection monitor mainly inserts instrumentation code to the fast interpreter.

One main problem for introspection is how to bridge the semantic gap between the original high-level logic of the application and the low-level execution process inside the Dalvik VM. The Dalvik VM manages adequate runtime application state such as name of Class/Object, which is very helpful for constructing semantic-rich events. Inside the interpreter there are bytecode interpretation handler, exception handler and many other events' handler. Insert instrumentation code into these handlers helps locate the corresponding events effectively. What's more, DIAS belongs to Dalvik VM and it forks with the Zygote process into different instance for each analyzed application. That is to say, DIAS produces unique instance to analyze every application. This characteristic greatly reduces the potential interference.

Notice that the introspection monitor's probe does not contain any analysis function, and is not responsible for reconstruct semantic-rich events. The only task for the probe is to monitor the execution process and export useful data via analysis interfaces, which is discussed in the following subsection.

C. Analysis Interfaces

To understand an application's execution on the Dalvik VM, analyzing the runtime state is necessary. As the application's code executes, every step of the execution produces various kinds of runtime information(e.g., register changing, memory accessing, arithmetic operation). The runtime information for analysis is well managed by DIAS and is provided through DIAS's analysis interfaces.

DIAS's basic philosophy is interface oriented analysis. The analysis interfaces formulate the runtime data accessing and application manipulation to decouple the analyzing task with the functionality of the Dalvik VM. Automated analysis can be done by programming with these interfaces. Also, different analysis modules are connected by the analysis interface.

Since DIAS's target is to analyze applications on Android thoroughly, It should record all the runtime information and perform advanced analyzing functions. However the code size of the introspection monitor should be limited to keep the Dalvik VM tight. So the implementation of complex analyzing functions cannot be placed into the introspection monitor. To this end, DIAS's analysis interfaces are designed into two categories. The first category is data-centric interfaces mainly provided by the introspection monitor directly. Since the introspection monitor basically does not perform any analysis work, it simply reads low-level information from the Dalvik VM(e.g., register values, operands of opcode, pointer of Object/String) and exports it with data-centric interfaces. The second category is logic-centric interfaces mainly provided by the analysis module of DIAS. These kinds of interfaces not only export high-level complex information such as String or Object, but also provide certain analysis functions to manipulate the analyzed application. For instance, an interface for extracting certain parameters of an API call is able to filter invoked procedure, and then retrieve the parameter automatically.

Similar to JVM Tool Interface[19] which is not supported by Dalvik VM, DIAS's analysis interface covers almost every aspect of Dalvik VM execution. And the set of DIAS's analysis interface is not fixed but extensible. As the new analysis module is added into this set, the functionality of DIAS is

extended.

D. Analysis Modules

DIAS's systematic methodology to achieve automated analysis is via interface manipulation and module developing. The analysis module of DIAS framework is responsible for performing advanced automated analyzing functions. To implement the analyzing function, an analysis module manipulates the analysis interfaces to collect runtime information first, and then define its own logic code to deal with the collected data. Finally, an analysis module's function is also defined as an analysis interface for other modules to use.

As Figure 2 shows, a typical analysis scenario is that one module may communicate with introspection monitor to access Objects such as *Location Object*, *SMS Object* or *password String Object*. As mentioned above, the introspection monitor is not responsible to reconstruct the semantic-rich Objects. It only exports interface for analysis module to access the low-level pointer of these Objects. Thus the analysis module must contain corresponding parsing code to acquire the useful information from the pointer. After a module is developed for encapsulating certain Object's pointer, it can also exports an interface for other module to access this Object directly without re-defining the parsing logic. In this manner the code reuse principle is assured. And the automated analysis is easy to be implemented by combining different modules.

E. DIAS Manager

One problem for DIAS's analysis is that each process is isolated by the sandbox[3]. Because DIAS instance belongs to application's Dalvik VM, many restrictions are introduced if cross application analysis is employed. One major problem is that the developed analysis tools can only access application's own resources(e.g. files at /data/data/packageName). Thus, to develop analysis modules, the developer should carefully check any permission-denied situation.

Generally it is not suggested to break the sandbox restriction for security consideration. If analysis module wants to use restricted resource or communicates with other application's DIAS instance, DIAS will manage this cross-application analysis behavior by introducing a global module. This global module, DIAS Manager, is an application level application to help fulfill such functions. The DIAS manager executes as an independent process which communicates with every instances of the DIAS. It uses local socket to communicate and it aggregates information from each process to make comprehensive cross-application analysis.

III. Implementation

A. Dalvik Modification

We implement DIAS on Dalvik VM of Android 4.0.x, 4.1.x and 4.2.x. The source code is written mainly in C++ and ARM Assembly. In Dalvik the fast interpreter we modified is optimized and each opcode is interpreted with a strict 64 bytes block of ASM code. So the modification is highly restricted and is inappropriate to insert complex instrumentation code into the interpreter. DIAS's solution is to link ASM with C functions. Our implementation only inserts a function call stub at the very beginning of every opcode's interpretation code.

```
/* File: armv6t2/OP_MOVE.S */
/* ------------------------------ */
    .balign 64
.L_OP_MOVE: /* 0x01 */
    /* for move, move-object, long-to-int */
    /* op vA, vB */
#if defined(DIAS)
    mov r0, r4        @ r0<- program counter
    mov r1, r5        @ r1<- Frame pointer
    mov r2, r6        @ r2<- Thread pointer
    BL monitor_mov    @ Insert Probe
#endif
    mov     r1, rINST, lsr #12
    ubfx    r0, rINST, #8, #4
    FETCH_ADVANCE_INST(1)
    GET_VREG(r2, r1)
    GET_INST_OPCODE(ip)
    SET_VREG(r2, r0)
    GOTO_OPCODE(ip)
```

The instruction `BL` *monitor_mov* directs the execution process to DIAS's probe function. The inserted probe functions like *monitor_mov* are implemented using C or C++ because it is easy to maintain. Some of the declaration of the probes are give below:

```
void monitor_mov
    ( u2 * pc, u4 * fp, Thread * self );
void monitor_object
    ( Method * m, Object * obj );
void monitor_reg
    ( RegOpType type, u4 * fp, u2 index );
void monitor_opcode
    ( u2 * pc, u4 * fp, Thread * self, Method * method );
```

DIAS's introspection monitor inserts these probes to relative part of the Dalvik VM. Then the information is passed through the parameters. The passed parameters are simply some pointers to crucial environment struct of the Dalvik VM. Inside the probe functions the analysis interface is implemented to accomplish more complicated analysis logic.

In addition, the Just-In-Time compiler was introduced to improve performance by compiling Dalvik instruction traces into native machine code[13]. However, we don't make use of the JIT technique for two reasons: First, to insert code into interpreter with JIT is complicated, and even Dalvik's debugging system still uses the portable interpreter without JIT. Second, developing and maintaining the interface with high level language(C/C++) is much easier than maintaining an assembly version with JIT. So DIAS's online analysis disables Android's JIT when monitoring.

B. Interface Definition

For DIAS's analysis interface, the analysis requirement is always changing and it is not necessary to fix the range of the interface. DIAS first provides a set of basic interfaces to meet the requirements of simple online analysis. They are:

- *GetOpcode*. This interface exports the current state's opcode type. The opcode's type is related to ceratin operation. For instance *invoke* and *return* reveal a method invoking.
- *GetMethod*. This interface returns the name of current executed *Method* in string form. Certain kinds of API calls can be filtered by name using this interface.

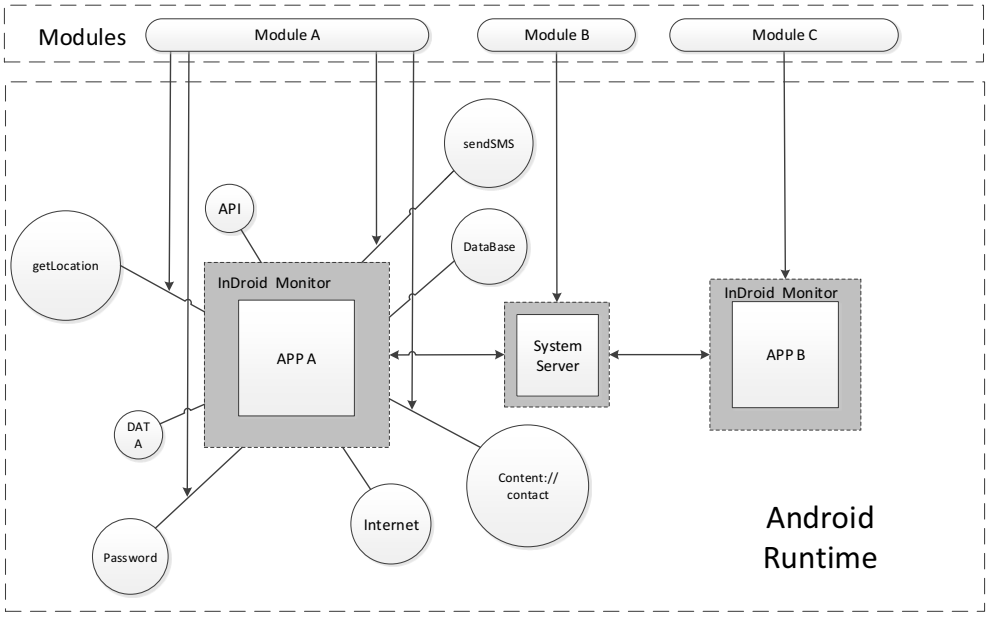

Fig. 2. Analysis based on DIAS

- *GetClass*. This interface returns the name of current executed Java *Class* in string form.
- *GetUID*. This interface returns current application's unique user ID(UID). The Android system assigns a UID to each Android application and runs it as that user in a separate process[4].
- *GetPID*. This interface returns current application's proccss ID(PID). Notice that Android allows developers set *android:process* so that components of different applications run in the same process.
- *GetInt*. This interface searches current state and returns integer for current operation(if exists). It may return null result. This interface is mainly used for monitor register changing.
- *GetString*. This interface searches current state and returns string for current instruction(if exists). Not every execution contains string, so it may return null result. Both crucial system parameters such as *ContentProvider URI* and privacy related data such as password can be monitored using this interface.
- *GetObject*. This interface searches current state and returns the pointer of a universal *Object*(except *String Object*) for current operation(if possible). It may also return null result. This interface is possibly the most powerful interface for the analysis module because most of the advanced analyses involve certain *Object* such as *android.content.Intent* and *android.telephony. SmsMessage*.
- *GetPackageInfo*. This interface returns current application's package information including package name, package signature and the private directory belonging to the application. This interface is the most complicated one exported by the introspection monitor. It seems that this interface should be implemented outside the introspection monitor. But the package information is somehow sensitive for outside modules to access, thus the monitor provides this interface directly.

After providing the basic interfaces, advanced analysis function is implemented with analysis modules. Figure 3 shows the interaction between analysis modules and simple interfaces provided by the introspection monitor. An analysis module is generally a C/C++ function which receives information from other interfaces and outputs wrapped information. In the following subsection we discuss some typical modules.

C. Deployment

The deployment of DIAS is simple. Because DIAS only modifies the Dalvik VM's library(/system/lib/libdvm.so) in system. So it just replaces the Dalvik VM library with an alternative one attached by the DIAS introspection monitor. For a device with the Android OS installed, this replacement scheme keeps all of the existing system and user's data. This feature solves the data lost problem of many low level analysis tools(such as TaintDroid) which need to flash images into the device.

To replace Dalvik VM(/system/lib/libdvm.so) the system partition should be mounted as read/write mode and this requires a root privilege. Although for many mobile devices the root process is applicable, the root privilege should be disabled to ensure the system's security model. Another choice

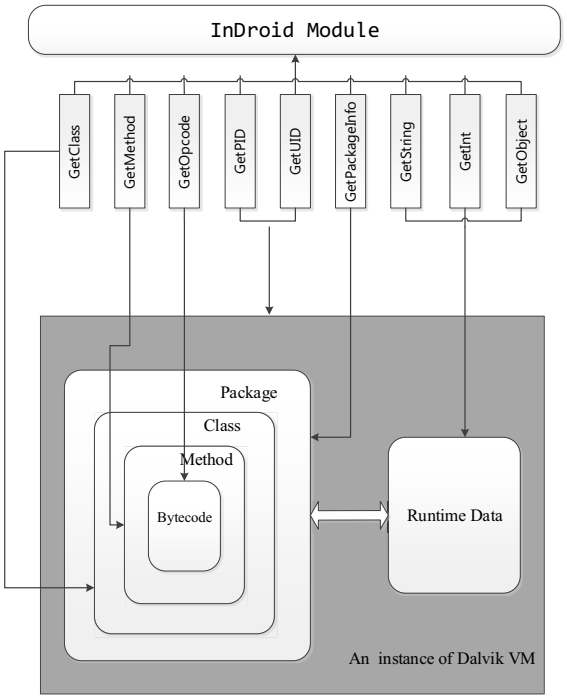

Fig. 3. Interaction between Analysis Modules and the Introspection Monitor

to install DIAS is to use recovery mode of the system, which is able to rewrite system partition's data. DIAS can be made as an update package and the user only need to flash this update package to finish deployment (Some devices checks the signature of manufacture, so a third party recovery tool is required to ignore this verification).

IV. Evaluation

A. Compatibility

DIAS is a universal and stable analyzing framework for various kinds of Android devices. We find that applying DIAS to different devices and different version of Android is applicable. We have implemented DIAS on major versions of the Android OS(4.0.x, 4.1.x and 4.2.x), deployed and tested DIAS on mobile phones and tablet (Samsung Galaxy Nexus, Samsung Galaxy S3, ASUS Nexus 7, Samsung Nexus S, Motorola Droid Razr). Among them, Samsung Galaxy Nexus and ASUS Nexus 7 are tested with both factory image and third party AOKP rom[5], Samsung Galaxy S3, Samsung Nexus S and Motorola Droid Razr uses original system provided by the manufacturer.

The fragmentation problems is a long term trouble for Android analysis tools. The compact architecture and Dalvik VM monitoring strategy of DIAS helps it adapt multiple environments. Manufacturer generally modifies original Android application layer to add new features such as launcher and custom applications. But the Android runtime is seldom modified. And the definition of Dalvik bytecode keeps stable for OS upgrading. Thus DIAS is able to work well with various Android OS versions from different manufacturers.

We developed analysis module based on DIAS to monitor SMS Manager, Contacts Manager and some other applications for online query(such as train and airplane schedule) simultaneously, record the sensitive APIs(e.g., SMS and contacts related APIs) and the string data into log file. All of the applications perform undisturbed with DIAS and the test discovers no compatibility problem. We also asked three users to test DIAS in daily use for about one month. The feedbacks are positive. All of the users response that the system is stable and normal operation is not affected. The feedback illustrates that DIAS keeps well user experience for it seldom interferes normal execution. As mainstream devices with improved processor more or less compensate the delay, we believe that DIAS is suitable for general and practical instrumentation.

B. Performance Overhead

The design of DIAS tries to keep well balance between performance and functionality. We use standard benchmark tools to quantify the overhead. We choose three benchmark tools: Linpack[9], BenchmarkPi[6] and caffeineMark[21]. These benchmark tools mainly focus on the performance on Dalvik VM. Because DIAS is for Dalvik VM introspection and the native code's execution is not affected. Thus we do not choose the benchmark for native code such as graphic rendering.

The test platform is a Samsung Galaxy Nexus mobile phone with the Android OS 4.2.1. The tested DIAS framework only enables Dalvik VM introspection and no complex analysis module is loaded. (If the analysis module performs concrete analysis, there will be an overhead related to the analysis task and is hard to be quantized). We run the benchmark with fast interpreter with JIT(which is Android's standard configuration), the portable interpreter(which is Android's debugging configuration) and fast interpreter with DIAS introspection monitor(with JIT disabled).

As the Table I shows, the portable interpreter is about five times slower than the fast interpreter with JIT in benchmark test (Although the actual gap for common applications with more human-device interaction may not reach five times). That is to say, when an application is debugged using Android's portable interpreter, there is a 5x slowdown. The fast interpreter with DIAS introspection monitor is generally 2x faster than the portable interpreter. The benchmark result shows that DIAS's performance is better than that of Android's debugging infrastructure. And the benchmark gives probably the maximum gap between DIAS and the fast interpreter with JIT because the benchmarks suite is often calculation intensive and JIT compilation may boost the performance. So we believe that DIAS gives an acceptable overhead to make analysis practical.

TABLE I
BENCHMARK RESULTS

Test item	Fast Interpreter with JIT	Portable Interpreter	Fast Interpreter with DIAS introspection monitor
Linpack single thread	44.263 MFLOPS	8.211 MFLOPS	13.989 MFLOPS
Linpack multi thread	77.834 MFLOPS	14.754 MFLOPS	24.499 MFLOPS
CaffeineMark Sieve	8445	1464	2211
CaffeineMark Loop	17127	1324	1948
CaffeineMark Logic	10540	1419	2345
CaffeineMark String	5837	4714	5599
CaffeineMark Float	7678	993	1753
CaffeineMark Method	5942	1562	2285
CaffeineMark Overall	8604	1649	2462
Benchmark-Pi	476 milisec	2021 milisec	1417 milisec

V. RELATED WORK

A. Android Debugging Infrastructure

The Android SDK provides a set of supportive components need to debug applications. The main components that comprise a typical Android debugging environment are:

1) Android Debug Bridge: The Android Debug Bridge(ADB)[1] acts as a middleman between a device and a development system. It provides various device management capabilities, including moving and syncing files to the emulator, running a shell on the device or emulator, and providing a general means to communicate with connected emulators and devices.

2) Dalvik Debug Monitor Server: The Dalvik Debug Monitor Server(DDMS)[7] is a graphical program that communicates with your devices through ADB. DDMS can capture screenshots, gather thread and stack information, spoof incoming calls and SMS messages, and has many other features.

3) JDWP debugger: The Dalvik VM supports the Java Debug Wire Protocol(JDWP) protocol[8] to allow debuggers to attach to a VM. Each application runs in a VM and exposes a unique port. A JDWP-compliant debugger can be attached to this port so it can communicate with the application VMs on devices.

4) JVM Tool Interface: The JVM Tool Interface(JVM TI)[19] is a programming interface which provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine. JVM TI is intended to provide a VM interface for the full breadth of tools that need access to VM state. Tools can be written directly to JVM TI or indirectly through higher level interfaces.

B. Android Application Online Analysis

Android's own debugging infrastructure provides *Dalvik Debug Monitor Service(DDMS)* and *Logging system*[2] to help analyzing. Although using *DDMS* could monitor the execution of applications. debugging is however not so automatic and the overhead for heavy instrumentation is high. And Debugger's interface is not extensible for designing new tools. The Android Logging system is to output four kinds of message. It does help analyzing applications. The problem is that it doesn't cover all the output runtime data. And for a heavy instrumentation the Logging system brings very bad performance. DIAS is a build-in analysis framework for the Dalvik VM and reduce the overhead of remote debugging's interaction. The analysis module of DIAS is able to perform complicated analysis logic automatically.

Dalvik VM currently does not support full features of the JVM TI. Chien-Wei Chang, et al. implemented a subset of JVM Tool Interface on Dalvik Virtual Machine[12]. However they did not implement fine-grained monitoring such as bytecode instrumentation. DIAS is able to fulfil fine-grained monitoring and analysis. The analysis interface of DIAS covers each bytecode's interpretation. What's more, the interfaces themselves are also extensible. Analyst can design new modules to extend the interfaces and the functionality of the framework.

Many online analysis tools for Android modify the analyzed application's code to insert instrumentation code. **Redexer**[20] is a bytecode instrumentation framework for Dalvik bytecode (used in Android applications). Redexer is a set of OCaml based utilities which helps programmers parse, manipulate, and generate bytecode for the Dalvik VM. Redexer modifies applications code to insert instrumentation code. [http://www.cs.umd.edu/projects/PL/redexer/] **Aurasium**[23] is a tool that can rewrite applications to intercept system calls to enforce security policies. **Dr. Android**[17] is also a bytecode rewriter for Android that removes Android permissions in existing application package and replaces them with a specified set of fine-grained versions. The major negative for these bytecode rewriting technique based tools is that the instrumentation code inserted into the application breaks the original signature of the application, and the modification of the application may affect the execution stability potentially. DIAS introspection monitors the Dalvik VM rather than insert instrumentation code into the application. DIAS adopts non-intrusive instrumentation style and doesn't modify application's code, thus doesn't break the signature of the application which is very important for verification.

There are also many tools which modify the system framework for analysis. **TaintDroid**[15] is a popular information flow tracking system for realtime privacy monitoring on Android smartphones. TaintDroid tracks the flow of sensitive information via Dalvik VM introspection, which is also adopted by our framework. TaintDroid tracks the propagation of tainted data from sensible sources(in program variables, les, and IPC) on the phone and detects unauthorized leak-

age of this data. However, TaintDroid is a specific purpose analysis framework, and its architecture is far from compact. **ProfileDroid**[22] is a comprehensive, multi-layer system for monitoring and profiling Android applications. It profile apps at four different layers. Although ProfileDroid is a systematic framework for application profiling, its architecture is complex and it does not support extended module well. Because it relies on many existing system components such as *adb* and *logcat* for characterizing Android applications behaviors at multiple layers. It cannot provide fine-grained analysis and better performance. **Droidscope**[24] is an Android analysis platform for malware analysis. It reconstructs both the OS-level and Java-level semantics simultaneously and seamlessly. DroidScope exports three tiered APIs that mirror the three levels of an Android device: hardware, OS and Dalvik Virtual Machine. One major disadvantage is that DroidScope requires extensive modification to the Android OS, which has significant usability issues and hinders efforts for widespread adoption. The architecture of DIAS is lightweight compared with DroidScope and is easy to be applied.

VI. Conclusion

In this paper we present DIAS, an automated online analysis framework for Android application. DIAS is designed to be open and extensible. It provides systematic methodology and adequate analysis interfaces to support automated application analysis, and it is convenient for automated analysis of performance, behavior and security verification of the application. New analysis module is encouraged to be added to extend the functionality of the framework. DIAS also controls the overhead in an acceptable range to guarantee that the applications could be executed normally.

References

[1] Adb (android debug bridge) - android developers. http://developer.android.com/tools/help/adb.html.

[2] Log — android developers. http://developer.android.com/reference/android/util/Log.html, .

[3] Permissions — android developers. http://developer.android.com/guide/topics/security/permissions.html, .

[4] Designing for security — android developers. http://developer.android.com/guide/practices/security.html, .

[5] Android open kang project. http://aokp.co.

[6] Benchmarkpi. http://androidbenchmark.com.

[7] Using ddms — android developers. http://developer.android.com/tools/debugging/ddms.html.

[8] Java debug wire protocol. http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html.

[9] Linpack for android. http://www.greenecomputing.com/apps/linpack/.

[10] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.

[11] D. Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

[12] C.-W. Chang, C.-Y. Lin, C.-T. King, Y.-F. Chung, and S.-Y. Tseng. Implementation of jvm tool interface on dalvik virtual machine. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 143–146. IEEE, 2010.

[13] B. Cheng and B. Buzbee. A jit compiler for android's dalvik vm. In *Google I/O developer conference*, 2010.

[14] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2008.

[15] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. 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*, pages 1–6, 2010.

[16] J. Gray. Why do computers stop and what can be done about it. In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. 2011.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[19] K. OHair and J. J. Heiss. The jvm tool interface (jvm ti): how vm agents work. *Web page, Dec*, 2006.

[20] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster. Application-centric security policies on unmodified android. *UCLA Computer Science Department, Tech. Rep*, 110017, 2011.

[21] R. Reddy. Caffeinemark. https://play.google.com/store/apps/details?id=com.android.cm3.

[22] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[23] R. Xu, H. Saıdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security*, 2012.

[24] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 29–29. USENIX Association, 2012.