WrapDroid: Flexible and Fine-Grained Scheme Towards Regulating Behaviors of Android Apps

Xueqiang Wang[1,2,3], Yuewu Wang[1,2], Limin Liu[1,2(✉)], Lingguang Lei[1,2], and Jiwu Jing[1,2]

[1] Data Assurance and Communication Security Research Center, CAS, Beijing, China
[2] Institute of Information Engineering, CAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
lmliu@is.ac.cn

Abstract. Accompanying the wide spread of Android mobile devices and the openness feature of Android ecosystem, untrusted Android apps are flooding into user's device and prepared to perform various unwanted operations stealthily. To better manage installed apps and secure mobile devices, Android app behaviour regulating schemes are required. In this paper, we present WrapDroid, a dynamic app behaviour regulating scheme on Android device. Different from other similar approaches, the key components of WrapDroid are implemented based on dynamic memory instrumentation and system call tracing and require no modification to Android system source code. Thus, WrapDroid could be flexibly adopted by Android devices. Moreover, by automatically reconstructing call context of Java or native operations, WrapDroid may provide a full range of control on both java runtime and system call layers of an app. We also develop a WrapDroid prototype and evaluate it on several devices from different mainstream OEMs. Evaluation results show that WrapDroid can effectively regulate the behaviors of Android apps according to given policies with negligible performance overhead.

Keywords: Android · App behaviour regulating · Dynamic instrumentation · Flexible · Fine-grained

1 Introduction

Google's Android is undoubtedly the most prevalent mobile platform in the world. In the second quarter of 2014, Android's market share of the global smartphone shipments reached record 84.6 % [1]. Unfortunately Android's growth in popularity and its openness feature of app ecosystem have also raised increasing security concerns. According to the report by Cisco, 99 % of all mobile malwares in 2013 targeted Android platform [2]. These malicious apps perform various harmful operations, such as reading personal information or sending SMS without the user's consent, which may incur privacy leakage or other losses for device user. While anti-virus measures have been adopted, it is still hard to ensure that all malicious code is isolated from user's Android device.

J. Lee and J. Kim (Eds.): ICISC 2014, LNCS 8949, pp. 255–268, 2015.
DOI: 10.1007/978-3-319-15943-0_16

Regulating the behaviors of untrusted Android apps according to given security policies may hold back the attack procedure of malicious code and secure user's device effectively. Android provides permission system to control app's behaviors. However, this permission model is too coarse-grained and only grants an "all-or-nothing" installation option for mobile users to either accept all the permissions an app asks for or simply decline to install the app. In Android 4.3, an experimental feature called App Ops [3] is added to permit mobile users to configure one app's runtime permissions, but this feature has been removed from Android 4.4.2 due to the increasing burden for user configuration and the impacts on advertisement market [4].

Several other works are carried out to exert more fine-grained controls on app's behaviors. These works mainly focused on two major research directions. The first direction is accomplished by modifying Android source code. Several extensions have been introduced into Android permission framework by system customization [5–9]. And some efforts are made to adopt mandatory access control (MAC) to Android [10,11]. However, these approaches require Android source code modification and would suffer from deployment problem. Changes on a general Android branch is hard to be built for devices of different OEMs because of their heterogeneities.

The second one is integrating behavior enforcement module into Android apps with app rewriting before the app is installed on user's device [12–18]. This approach does not require any change to Android source code and easy to deploy. However, it also has several limitations. Firstly, user have to ensure that all installed untrusted apps are correctly wrapped. Thus, this scheme applies only to ordinary apps because those stock apps cannot be replaced easily and the burden on user is greater. Secondly, the scheme, based on rewriting bytecode of an app, can be easily bypassed by dynamically loading native code into app's address space. Finally, the impaction on apps is permanent. For instance, the data associated with the original app will be lost because of different app signature.

In this paper, a flexible and fine-grained scheme called WrapDroid is presented to regulate Android apps' behaviors dynamically. We observed that even through an app may realize an operation through various interfaces, its operation traces will always appear in Dalvik interpreter or go through Linux system calls. By checking these two points, app's behaviours, whether performed by native code or Java reflections, can be examined and supervised completely. Fine-grained policies to constrain behavior of Android apps are supported by our approach because parameter details of each operation are analysed automatically at the check points. To guarantee its flexibility and easy deployment, the enforcement modules are based on dynamic Dalvik instrumentation and system call tracing. The advantages of the dynamic implementation are threefold. Firstly, both ordinary apps and stock apps are under regulation. Secondly, the modules that loaded during run-time are also monitored. Finally, we can launch behavior regularization flexibly by instrumenting app's memory address space and remove its effect by restoring context of the app. In other words, app's running environment are modified temporary, which brings no further impact

on original app. This paper mainly focuses on the implementation of behavior regulating, so only an example policy is discussed even through various security policies can be accepted by WrapDroid. Moreover, Java code regulating can also be achieved in ART runtime by *.oat* instrumentation, which makes WrapDroid available to the latest Android version with some amending.

In summary, we make the following contributions in this paper.

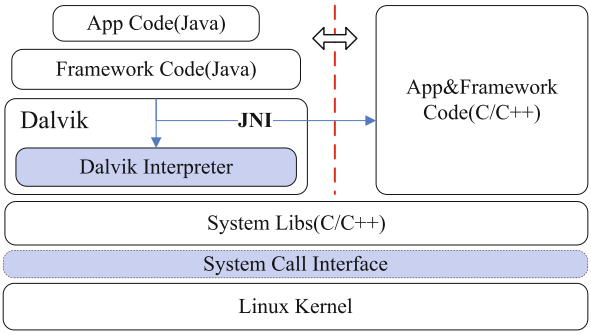

Fig. 1. Android application code structure

- We solved the flexibility problem of regulating behaviors of Android apps. Our approach is based on dynamic Dalvik instrumentation and system call tracing, which requires no modification to Android OS. Moreover, the impact of WrapDroid is minimized because only address space of supervised apps are instrumented temporarily.
- We achieve a complete and fine-grained control on Android apps. Based on monitoring of runtime interpreter and system calls of supervised app, both Java and native code behaviors are put under monitoring. These behavior context accompanying with well-designed policies make our scheme fine-grained.
- We develop a WrapDroid prototype and evaluate its effectiveness and efficiency on several Android devices. The evaluation result show that WrapDroid can effectively regulating behaviors of Android apps and meanwhile incurs an ignorable performance overhead.

The remaining of the paper is organized as follows. Section 2 introduces necessary background knowledge. Section 3 presents WrapDroid system design. A prototype implementation is detailed in Sect. 4. Section 5 discusses the evaluation of WrapDroid. We describe related works in Sect. 6. Finally, we conclude the paper in Sect. 7.

2 Background

2.1 App Code Structure

Android, built on top of Linux Kernel, adopts a unique app architecture that supports both Java code and native (C/C++) code. For one thing, Java code is

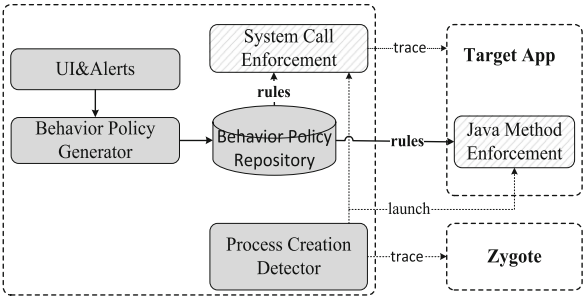

Fig. 2. WrapDroid architecture

compiled into bytecode and runs on register-based Dalvik VM. For another, linux shared objects (.so) are available for apps to reuse native libraries or accelerate performance-critical tasks. These two parts interacts with each other based on Java Native Interface (JNI) specification and reflection. Figure 1 shows in detail the code layers in an app's address space.

App Java code is written based on well-documented Android framework APIs, most of which are implemented in Java. In order to get executed, Java classes are compiled into Dalvik Executable Format (.dex) and transferred to Dalvik VM. The VM is introduced into app's address space as *libdvm.so* and all Java methods are eventually managed in its interpreter. However, part of the Java methods of an app or Android framework are declared as JNI methods and their required operations, like file and socket operations, are completed in native code. The native code relies on lower level system libraries and system calls of Linux kernel.

No matter how complicated an app seems, its code will always appear in Dalvik interpreter or go through related system calls in order to realize its functionalities. This provides a convenient and complete check point at which an app's behaviours can be dynamically examined and supervised.

2.2 App Launching

In Android, app processes are not generated by forking an ordinary process directly as that in Linux. Instead, *zygote* process is created during system booting to serve all process creation request. After we tapped an app icon on the home screen, a trusted system service, named Activity Manager Service would send a process creation request to *zygote* through zygote socket under the hood. The *zygote* is able to fork itself and configure child process properties (e.g. gid, uid) according to the request. Because the child process has inherited loaded libraries, resources and a Dalvik instance from *zygote*, it is prepared to load Java classes of the specific app and get them executed. Therefore, invocations of *fork* system call by *zygote* indicate the very beginning of app processes' life cycle. They work as reliable trigger event at which an app is put under monitor.

3 System Design

As described in Sect. 2.1, by monitoring interpretation of bytecode or invocation of system calls initiated by an app, a complete behavior map of the app can be retrieved and supervised. Our WrapDroid is designed based on the above observation. The WrapDroid is composed of six components that scatter in different processes and cooperate to supervise target apps. Its architecture is shown in Fig. 2.

It is device owner oriented and the *UI&Alerts* accepts behaviour policy items from the owner. These items consist of target apps and a set of regulated behaviour patterns for them. The *Behaviour Policy Generator (BPG)* translates all the items and passes them on to *Behaviour Policy Repository (BPR)*.

When WrapDroid is deployed, the *Process Creation Detector (PCD)* starts tracing *zygote* for newly created processes. The *PCD* maps these child processes to their hosting apps. If a hosting app is designated as a monitoring target, its processes are put under monitor immediately after process creation completes. The *System Call Enforcement (SCE)* implements a system call interposition based on *ptrace* and is responsible to regulate native code operations of target process by enforcing confinement on system calls according to *BPR*. The *Java Method Enforcement (JME)* is injected into target process and works by hijacking entry functions of interpreter. Every time a sensitive method frame is retrieved from interpretation stack, *JME* would determine whether its execution should be confined.

4 Implementation

We have implemented a WrapDroid prototype and the detailed description of each key component is given below.

4.1 Process Creation Detector

As shown in Sect. 2, *zygote* is responsible to fork an app process upon a process creation request from Activity Manager Service. To guarantee that the entire life cycle of the target app is monitored, *PCD* is introduced to detect newly created processes by tracing system calls of *zygote*.

We surveyed several existing system call tracing techniques, but most of them cannot satisfy the flexibility requirement. Some of them require enabling certain Linux kernel features, including *kprobes* [19]. Some others, like *ftrace* [20], is inflexible and weak in system call supervision. In our work, we monitor system calls of target process based on *ptrace*. Because signals are delivered from target process to our tracer at both the entry and exit of system calls, we have to use a history stack to distinguish between *syscall-enter-stop* and *syscall-exit-stop*. Moreover, by analysing and interposing registers and address space of the target process, detailed parameters of system calls are retrieved and their execution controlled.

Because any app identifier hasn't been set for the new process when *fork* returns, we have to relate the process to an app through other ways. In Android, each app package is regarded as a user and assigned a unique *uid*. We observed that the new process will set its *uid* right after *fork* returns from *zygote*. Therefore, identifying parameters of *setuid* helps to map the process to an app. If the hosting app should be monitored, WrapDroid would immediately start *SCE* and inject *JME* component into the process.

However, not all processes that related to an app derive from *zygote*. For example, an app process may launch *Runtime.exec* or directly *fork* its own child process. In this case, it's not complete to trace only *zygote* in *PCD*. To address this problem, the app process tree is traced recursively in *PCD* by means of specifying PTRAC_O_TRACEFORK option for *ptrace*.

4.2 Java Method Enforcement

The *JME* works in target process and is based on code injection. Because Android is built on top of Linux kernel, code injection techniques used in Linux also apply to Android. And *ptrace* offers us an available way. Listing 1.1 shows an overview of code injection procedure. Target process is attached to our tracer by *attchToTarget*. Upon a successful attachment, registers of target process are reserved so as to restore original execution state. Then we can obtain free memory for injected code by initiating a *mmap* function in target process. Because all Android processes share an identical mapping of system libraries including *libc.so* where *mmap* is defined, the *mmap* address in target process is the same as that in the tracer and can be easily obtained. After running *putCodeInTarget* method, the free memory is filled with the injected code. By adjusting registers (*pc, lr, etc.*) of target process, the injected code gets executed before restoring normal code sequence. Now the environment of target process has been changed and we can detach our tracer.

Listing 1.1. Code injection based on *ptrace*

```
1  void codeInjection(pid_t pid,
                      const char* func,
3                     int length,
                      ...){
5      attchToTarget(pid);
       regs = getRegsOfTarget(pid);
7
       //freeMem: where code is injected
9      freeMem = getMemoryOfTarget(pid);

11     //func: address of injected code
       //length: size of the code
13     putCodeInTarget(pid, freeMem,
                       func, length);
15
       manageRegs(&regs);
17     setRegsOfTarget(pid, regs);
       detachTarget(pid);
19 }
```

Every Dalvik thread maintains an interp stack in order to manage Java method and mimicked native method frames. These frames could be consumed by interpreter or by execution of native functions. Figure 3 shows some details inside of the VM. After obtaining a frame from interp stack, the thread would firstly decide whether the frame represents a JNI method. Native code of a JNI method is retrieved from *DalvikBridgeFunc* field of *Method* structure. A Java method frame is dispatched to different entries of interpreter in the light of different interpreter mode. The interpreter in fast/jit mode is coded in assembly language or has adopted Just In Time (jit), and thus relatively faster in bytecode interpretation. The method frame is delivered to an entry function named *dvmMterpStdRun* under this mode. However, for portability consideration, interpreter in portable mode is implemented in C language. Its entry function is *dvmInterpStd*. In our work, interpreter mode is fetched from system property *dalvik.vm.execution-mode*. By inline-hooking of the above entry functions, execution flow can be regulated based on information of current method and thread state. As can be seen in Fig. 3, function *javaMethEnforce* hijacks the interpreter and is responsible to enforce constraints on Java methods.

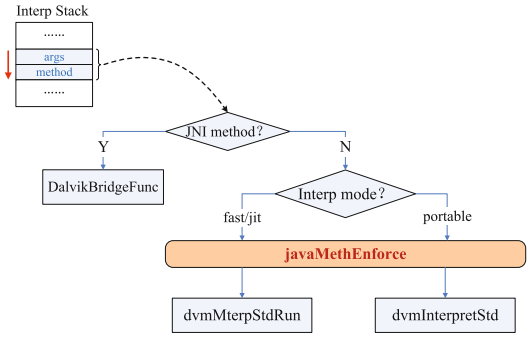

Fig. 3. Java method enforcement implementation

The key step of *javaMethEnforce* is extracting method details. We process parameters by scanning the method signature, where parameter types are designated. Primitive parameters are directly stored in stack and reference ones could be retrieved from Dalvik heap. Some simple reference variables, including *String* and primitive arrays, are parsed directly from heap memory. While the compound ones, like *Intent*, are sophisticated and made up of nested variables. Inside of Dalvik, every reference type is represented by a *ClassObject*. All fields and methods of the type are defined in the *ClassObject*. We automatically analyse compound variables by enumerating its fields and reduce them to simple types. To make our idea clear, a typical method that containing a SMS intent is given in Table 1. Particularly, reflection method can be detected from an object that labelled *Ljava/lang/reflect/Method;*.

Table 1. A typical method resolved in *JME*

Property name	Property value
Package	com.example.android.Msg
Thread	Main
Class	Landroid/app/instrumentation;
Method	checkStartActivityResult(IL)
Argument	Intent[mAction(android.intent.action.SEND), Uri(smsto:10010)]

Java method can be regulated by fabricating input parameters and return values. However, implementation of *JME* depends on interleaved structure and function definitions inside Dalvik. It's tedious to include all the definitions one by one. Hence, we compile *JME* on top of Dalvik part of Android Open Source Project (AOSP). Because no hardware module is involved in Dalvik, the recompiled *JME* applies to devices from different OEMs. Moreover, there exist different definition details through Android versions. For example, interpreter entry of Android 2.3 requires parameter of type *InterpState* pointer, while Android 4.0 or higher version requires *Thread* pointer. To guarantee compatibility, we customize a *JME* module for every existing Android version.

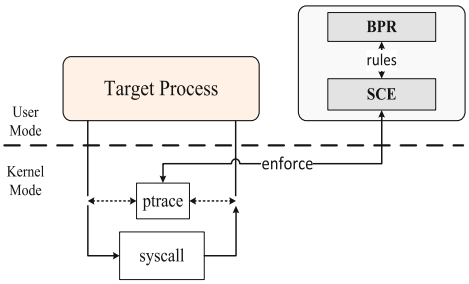

Fig. 4. Workflow of system call enforcement (SCE)

4.3 System Call Enforcement

The *SCE* component implementes the *ptrace* approach described in Sect. 4.1. Figure 4 outlines how *SCE* works. The target process would be suspended at the entry and exit of system calls. And meanwhile, *SCE* would be awakened by signals from the target process. The *SCE* is then able to enforce constraints based on system call details and rules from *BPR*.

For example, *SCE* enforces a series of network rules to regulate how an app accesses network resources. Basically, each app is prohibited from interacting with malicious remote addresses that defined in an IP blacklist. By managing

parameters at the entry of socket related system calls, including *connect*, *sendto* and *recvfrom*, communications between the app and malicious network servers are restricted. Furthermore, to fully regulate an app's network access and meanwhile guarantee its usability, network data that from untrusted IP are forged before handling it to user space app. It is accomplished by modifying return data at the exit of socket calls, like buffer of *recvfrom*.

4.4 Behaviour Policy Definition

In WrapDroid, device user is able to initiate policies for an app with *UI&Alerts* module. The *BPG* is responsible to translate human-readable policies to policy items stored in *BPR*. We have defined WrapDroid policy language on top of policies of FireDroid [21] and the syntax is shown in Listing 1.2. Compared to FireDroid, our system pays more emphasis on high-level Java code regulation and alleviates the tediousness of defining low-level policies.

Listing 1.2. The syntax of the WrapDroid policy language

```
1 Package Operation [param-list]
2 if condition then outcome
```

The *package* works as a unique identification for target processes and is the basic unit for policy enforcement. A *package* requests an *Operation* on a *condition*. The policy would evaluate the *outcome* (allow or deny) for this behaviour. Different from FireDroid, our *condition* clause supports not only related context information but also how this behaviour is performed (e.g., normal Java APIs, Java reflections, native code). To make the syntax clear, we introduce a scenario to regulate SMS behaviours. Listing 1.3 shows how SMS destination, frequency and content are constrained for package *com.android.mms*. Particularly, some malicious apps may send SMS in native code by reflection to evade static analysis. The *isByReflection* clause prevents this kind of behavior.

Listing 1.3. Policy to control SMS

```
1 com.android.mms sendSMS [dst, content, lastSMSTime]
2   if (isByReflection) then deny
3   if (blacklist contains dst) then deny
4   if ((currentTime-lastSMSTime) < 1h) then deny
5   if (content contains 'Y') then deny
6   if (content contains number) then deny
```

5 Performance Evaluation

We evaluate the performance of WrapDroid in three aspects: (I) to demonstrate its effectiveness by enforcing restriction on prevalent apps; (II) to evaluate app performance overhead caused by adoption of WrapDroid; (III) to evaluate the impact on app launching. The experiment results demonstrate that WrapDroid can effectively regulate app's operations with reasonable performance overhead.

5.1 Effectiveness Evaluation

To test effectiveness of WrapDroid, we downloaded top 1050 apps from *wandoujia* [22], one of the most popular Android market of China, as an experiment sample. We evaluate how WrapDroid works to regulate SMS sending behavior of sample apps. By static analysis, 113 out of 1050 apps are found to request SEND_SMS permission in their manifest files. We originally planned to run each of the selected apps with *monkey*, an automatic event generator. However, only a few SMS sending operations are detected for random fuzzing feature of *monkey*. Hence, manual work is merged in our evaluation. Among the 113 apps, 35 apps have actually sent SMS and 40 destination numbers have been detected. In the experiment, we set the policy as "shutting down all the SMS message sending operations" and "restricting all SMS messages sent to "1065*"" separately. According to the bill from SMS service provider, WrapDroid meets our behaviour regulating expectation exactly.

5.2 App Running Efficiency

CaffeineMark 3.0 supports Android platform and runs as an Android app. It's score represents app's running efficiency. We run the benchmark on Nexus S with Android 4.1.2 under normal Android system and when WrapDroid is active. The result is shown in Fig. 5. As can be seen, the overhead of sieve, logic, loop and float tests incurred by WrapDroid is limited to 7 %. The string and method tests suffer from more performance loss of 16 % and 11 % because much more work is done on method and parameter analysis. While the overall overhead of CaffeineMark is 8.5 %, which means no noticeable impact has been brought to user experience.

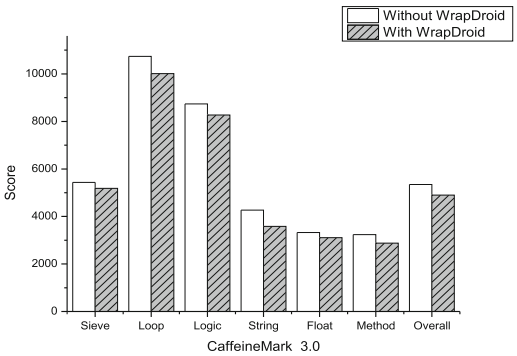

Fig. 5. CaffeineMark result of app running efficiency

Most of the functionality of Android apps are achieved by Framework APIs, which is not the key test factor of the above benchmark. Hence, we experimented on an app that performs a large amount of API invocations. An overhead comparison is made between WrapDroid and Aurasium [15], which is a

Table 2. Comparing WrapDroid with Aurasium

200 API invocations	Without WrapDroid	With WrapDroid	WrapDroid overhead	Aurasium overhead
Get device info	131 ms	145 ms	11 %	35 %
Get last location	71 ms	79 ms	12 %	34 %
Query contact list	132 ms	143 ms	8 %	14 %

policy enforcement scheme based on libc interposition. The evaluation result is shown in Table 2. Our WrapDroid is accomplished in higher Dalvik layer and thus more direct in parsing operations than Aurasium. Correspondingly, API invocation overhead of WrapDroid is much smaller than Aurasium.

5.3 App Launching Efficiency

As described in Sects. 2 and 4, app launching procedure stretches across several processes (e.g., *zygote*, *system_server*). Efficiency of app launching would be affected because these processes are under monitoring of system call tracing and runtime instrumentation. We evaluate it by designing an experimental app. The app initiates a *startService* request, and calculates time consumed when the service succeeds in running *onCreate*. The app is executed 10 times in normal Android and when WrapDroid is active on three devices: Nexus S (Samsung) with Android 2.3.6, Sony LT29i with Android 4.1.2, Meizu MX II with Android 4.2.1 and Samsung Galaxy Note II with Android 4.2.1. The result is shown in Table 3. We notice that the maximal time difference is within 23 ms, which can be ignored given the low frequency of app launching events.

Table 3. App launching efficiency

Device	Without WrapDroid	With WrapDroid	Delay
Nexus S (Samsung)	73 ms	90 ms	17 ms
Sony LT29i	95 ms	112 ms	17 ms
Meizu MX II	98 ms	106 ms	8 ms
Samsung Galaxy Note II	50 ms	73 ms	23 ms

6 Related Work

Researchers have worked on various aspects to regulate behaviors of Android apps and we categorize them into three classes by technical features.

Modifying Android source code. Many approaches based on source code customization have been proposed to regulate behaviors of Android apps. Some

of them aim to enforce constraints based on extending permission mechanism. Apex [5] enables user to grant a selected set of permissions and supports user-defined restrictions on apps. CRePE [6] introduces external device context to refine permission policies. The privacy mode of TISSA [7] empowers users to define the kinds of personal information that are accessible to apps. And Comapat [8] restricts permissions of components to mitigate security problems aroused by a third-party component. Another way is accomplished by introducing Security Enhanced Linux (SELinux). References [10,11] implements concepts of SELinux on both Android's middleware and kernel layers to enhance a flexible mandatory access control (MAC). Besides directly regulating behaviors of Android apps, securing privacy data leads to the same destination. Reference [23] replaces private data with dummy data before providing it to apps. However, these approaches require Android source code modification and would suffer from deployment problem because of vendor customization. Our system performs all modification to Android apps by dynamic Dalvik instrumentation and system call tracing and thus can be deployed easily.

App rewriting. To make them portable, many approaches are implemented by integrating behavior regularization modules into Android apps by rewriting. With the rewritten dalvik bytecode, [12] is able to identify and interpose Security Sensitive APIs. Reference [13] uses static and dynamic method interception to retrofit app's behaviors. Reference [14] is an on-the-phone instrumentation scheme and its policies are based on interception of high-level java calls. Nevertheless, security policies of [15] are enforced by interposing low-level *libc.so*. Reference [16] introduces a new module that supports parameterized permissions and requests of sensitive resources from apps are forwarded to this module. App rewriting is an effective way that requires no modification to Android ROM. However, incomplete implementations of bytecode rewriting may result in several potential attacks [17]. It is difficult to assure that all apps installed are rewritten version of the original app. Due to signature difference of repackaging process, all history information of the original app cannot be shared by the rewritten app. In addition, system apps cannot be replaced easily so app rewriting only applies to third-party apps.

7 Conclusions and Future Work

Based on an observation that behaviours of an app can always be monitored completely in Dalvik interpreter or execution of system calls, we propose an app behaviour regulating scheme named WrapDroid based on dynamic instrumentation of Dalvik runtime and system call tracing. WrapDroid is flexible to enforce constraints on Android apps because this dynamic approach requires not any Android source code modification. WrapDroid monitors app's behavior from both Dalvik and system call layer, so operations of Java and native code are regulated completely. Moreover, through automatic recovery of operation context (e.g. parameters or call logs) and a set of policies, we can achieve a fine-grained control on Android apps. The evaluation of WrapDroid prototype demonstrates

that our system can effectively regulate apps' behaviors with reasonable overhead.

At Google I/O 2014 conference, Android L was unveiled and the previously experimental Android Runtime (ART) has replaced Dalvik as a default environment. ART compiles byte code into executable ELF only once during app installation. We are now designing a scheme that monitors an app executed on ART by instrumenting compiled *.oat* executable file. WrapDroid would be fully effective on even the newest ART runtime after this future work is merged in.

Acknowledgement. This research was supported by the National Grand Fundamental Research 973 Program of China (Grant No. 2013CB338001 and No. 2014CB340603) and program of Computer Network Information Center of Chinese Academy of Sciences.

References

1. Strategy analytics: 85 % of phones shipped last quarter run android. http://bgr.com/2014/07/31/android-vs-ios-vs-windows-phone-vs-blackberry/
2. Cisco 2014 annual security report. http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf
3. App ops: Android 4.3's hidden app permission manager, control permissions for individual apps! http://www.androidpolice.com/2013/07/25/app-ops-android-4--3s/hidden-app-permission-manager/-control-permissions-for/-individual-apps/
4. App ops removed by google in android 4.4.2 update. http://www.phonearena.com/news/App-Ops-removed-by-Google-in-Android-4.4.2-update_id50340/
5. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (2010)
6. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: context-related policy enforcement for android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 331–345. Springer, Heidelberg (2011)
7. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)
8. Wang, Y., Hariharan, S., Zhao, C., Liu, J., Du, W.: Compac: enforce component-level access control in android. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (2014)
9. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in android. In: Annual Computer Security Applications Conference (2009)
10. Bugiel, S., Heuser, S., Sadegh, A.R.: Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In: 22nd USENIX Security Symposium (USENIX Security 2013) (2013)
11. Smalley, S., Craig, R.: Security enhanced (SE) android: bringing flexible mac to android. In: NDSS (2013)
12. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-arm-droid: a rewriting framework for in-app reference monitors for android applications. In: Proceedings of the Mobile Security Technologies 2012, MOST 2012. IEEE (2012)

13. Davis, B., Chen, H.: RetroSkeleton: retrofitting android apps. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (2013)
14. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard – enforcing user requirements on android apps. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 543–548. Springer, Heidelberg (2013)
15. Xu, R., Saïdi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium (2012)
16. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (2012)
17. Hao, H., Singh, V., Du, W.: On the effectiveness of API-level access control using bytecode rewriting in android. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (2013)
18. Hao, S., Li, D., Halfond, W.G., Govindan, R.: SIF: a selective instrumentation framework for mobile applications. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (2013)
19. Kernel probes. http://sourceware.org/systemtap/kprobes/
20. Ftrace. http://elinux.org/Ftrace
21. Russello, G., Jimenez, A.B., Naderi, H., van der Mark, W.: FireDroid: hardening security in almost-stock android. In: Proceedings of the 29th Annual Computer Security Applications Conference (2013)
22. Wandoujia. http://www.wandoujia.com/
23. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011 (2011)