Reference Hijacking: Patching, Protecting and Analyzing on Unmodified and Non-Rooted Android Devices

Wei You[1,2], Bin Liang[1,2,*], Wenchang Shi[1,2], Shuyang Zhu[1,2], Peng Wang[1,2], Sikefu Xie[1,2] Xiangyu Zhang[3]

[1]Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, Beijing, China
[2]School of Information, Renmin University of China, Beijing, China
{youwei, liangb, wenchang, zhushuyang, pengwang, decadall}@ruc.edu.cn

[3]Department of Computer Science, Purdue University, Indiana, USA
xyzhang@cs.purdue.edu

ABSTRACT

Many efforts have been paid to enhance the security of Android. However, less attention has been given to how to practically adopt the enhancements on off-the-shelf devices. In particular, securing Android devices often requires modifying their write-protected underlying system component files (especially the system libraries) by flashing or rooting devices, which is unacceptable in many realistic cases. In this paper, a novel technique, called *reference hijacking*, is presented to address the problem. By introducing a specially designed reset procedure, a new execution environment is constructed for the target application, in which the reference to the underlying system libraries will be redirected to the security-enhanced alternatives. The technique can be applicable to both the Dalvik and Android Runtime (ART) environments and to almost all mainstream Android versions (2.x to 5.x). To demonstrate the capability of reference hijacking, we develop three prototype systems, *PatchMan*, *ControlMan*, and *TaintMan*, to enforce specific security enhancements, involving patching vulnerabilities, protecting inter-component communications, and performing dynamic taint analysis for the target application. These three prototypes have been successfully deployed on a number of popular Android devices from different manufacturers, without modifying the underlying system. The evaluation results show that they are effective and do not introduce noticeable overhead. They strongly support that reference hijacking can substantially improve the practicability of many security enhancement efforts for Android.

CCS Concepts

• Systems security → Operating systems security → Mobile platform security.

Keywords

Android; Security enhancement; Practicability

1. INTRODUCTION

Android has become the most widely used mobile operating system. It dominated the global smartphone market with an 83% share in Q2 of 2015 and continues to grow steadily [21]. At the same time, people become increasingly dependent on handheld devices for processing personal information and handling business affairs. To guarantee the security of devices is of great importance for end users.

Over the past few years, many studies have been proposed to improve the security of Android, such as patching security vulnerabilities [43, 50], extending the standard security model [44, 54], and analyzing malicious applications [38, 41]. These studies have brought great improvement to the security of Android. However, less attention has been given to the *deployability* of these security enhancements, which has a significant impact on their adoption.

In fact, deploying security enhancements on off-the-shelf devices is not a trivial task. It is inevitable to modify the system component files of the Android OS. In practice, when patching a vulnerable system library, we need to replace it with a fixed alternative or rewrite it with patch code [43, 50]. Similarly, to enforce a customized security policy, the additional security module often needs to be introduced into the system libraries of the Android framework [32, 37, 44, 54]. For analyzing malware, the analyst also needs to effectively inspect the execution of the system libraries in addition to the target applications themselves [38, 41, 52]. However, Android has more stringent security restrictions than desktop platforms. The underlying system libraries are stored in special folders, which are only writable for the *root* user. In Android, root privilege is not granted to end users to avoid privilege abuse. This poses an obstacle for deploying security enhancements. Without root privilege, it is impossible to rewrite or replace the original system libraries.

To deploy security enhancements, most exiting studies employ the following two ways. The first way (employed by [38, 44]) is to integrate the Android source tree with security enforcement code, and compile the modified source code to a new system image for flashing into devices. It involves a complex series of steps, and is at the risk of bricking devices. The second way (employed by [43, 50]) is to root devices and inject security enforcement code into the target application process. It compromises system integrity, and may cause severe hazard if the root privilege is abused [25]. In practice, these two ways are not acceptable for normal users. This substantially hinders the widespread adoption of many security enhancements.

It should be noted that a few attempts have been made to enforce *application-wide* security features without flashing or rooting devices. Aurasium [51] enforces a fine-grained permission policy. The global offset table (GOT) of the target application process is rewritten, such that calls to critical *libc* functions can be intercepted and validated. Boxify [30] proposes a novel technique to en-

* Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884863

force privilege separation policies. Untrusted applications are securely encapsulated in an isolated sandbox, such that inter-process communications and system calls of the untrusted applications can be mediated. Although these two approaches do not require modification of the underlying system, their capabilities are limited to enforcing function-call-level protection policies. It is very difficult, if not impossible, to enforce an in-depth security enhancement only by intercepting calls, such as completely repairing a vulnerable system library or tracking the execution of an underlying function. Based on the above discussion, we can see that *how to practically enforce in-depth security enhancements in the underlying system libraries without flashing or rooting devices remains an important open challenge.*

Our Approach. In this paper, we propose a novel technique, called *reference hijacking*, to conveniently introduce additional in-depth security features into underlying system libraries. Reference hijacking is mainly archived by a specially designed environment reset procedure triggered when the target application is launched. The reset procedure reconstructs a new execution environment for the target application, such that the system libraries can be loaded from a configurable location instead of the default folders. As a result, the reference of the target application to the underlying system libraries can be redirected to the security-enhanced alternatives. In theory, the technique can be regarded as an *inline reference monitoring* (IRM) [39] based approach. Compared with the existing IRM-based approaches [42, 51], our technique enables us to take full control over the reference of the target application to the underlying system libraries, rather than merely intercepting certain API calls. We evaluate the technique on prevalent off-the-shelf devices, involving almost all mainstream Android versions from 2.x to 5.x, and covering both the Dalvik and Android Runtime (ART) environments. The evaluation results show that the technique works well on all tested devices, and only introduces small launch time delay (detailed in Section 2).

To further demonstrate the capability of reference hijacking, we design and implement three prototype systems based on it for enforcing specific security enhancements to address three hot and important security issues of Android. The experiment results show that all of these three prototypes are effective and efficient, and can be directly deployed on the off-the-shelf Android devices.

The first prototype, *PatchMan*, is to patch vulnerable underlying system libraries. Vulnerabilities pose serious security threats and should be patched as timely as possible. Unfortunately, Google adopts a controversial patch policy. Patches are provided preferentially to the latest versions, while back-porting to the legacy versions is not promised and even refused [18]. Besides, many device manufactures only provide a limited support time window. Beyond the window, no further update will be provided even though those old devices are still widely used [23]. As a result, a large portion of devices in use contain critical vulnerabilities [13, 20]. PatchMan provides a third-party patch mechanism for poorly-supported devices. It allows organizations or end users to proactively fix vulnerable devices by themselves. By leveraging reference hijacking, the target application can conveniently use the patched system libraries as the substitute of the vulnerable ones (detailed in Section 3).

The second prototype, *ControlMan*, is to enforce a flexible access control policy on inter-component communications (ICCs). By default, Android enforces a permission-based security model to protect the ICC interfaces. However, this generic model is not completely suitable for some particular requirements. For example, a user may want to restrict ICCs of a target application, only allowing the applications in the white-list to send/receive messages to/from the target application. This requirement is reasonable, but beyond the capability of the standard Android security model. ControlMan provides a more flexible access control model, which enables end users to regulate the incoming and outgoing ICC behaviors of a target application by specifying customized security policies. The access control model is implemented in the enhanced system libraries, which will be used to replace the original libraries for the target application via reference hijacking (detailed in Section 4).

The third prototype, *TaintMan*, is to perform dynamic taint analysis (DTA) [48] for malware detection. Due to the openness of application markets and the lack of effective vetting process, Android accounts for 97% of all mobile malware [19]. Even in the official market, there still exist a considerable number of malware [33, 55]. Moreover, sophisticated adversaries have begun to employ anti-analysis technique to evade detection. Especially, some malware will deactivate their malicious logic if they perceive that they are executed in a monitored environment (such as an instrumented emulator) [49]. TaintMan enables analysts to analyze malware directly on a real device instead of an emulator, for capturing the hidden malicious behaviors. In TaintMan, taint tracking code is instrumented into both the target application and the underlying system libraries. With the help of reference hijacking, sensitive information flow involving the system libraries can also be effectively tracked and analyzed (detailed in Section 5).

These three prototypes have successfully demonstrated that reference hijacking does provide a totally effective solution to enforcing in-depth security enhancements for applications of interest on off-the-shelf devices. We believe that this study has made an important step to improve the practicability of a large body of security enhancement efforts for Android.

In summary, our two main contributions are the following.

- We present reference hijacking, a practicable technique for introducing in-depth security enhancements without flashing or rooting Android devices. It has been shown to work well on both the Dalvik and ART runtime environments and on almost all mainstream Android versions (2.x to 5.x).
- We develop three prototype systems to enforce specific security enhancements based on reference hijacking, involving patching vulnerabilities, enforcing ICC access control, and performing DTA analysis for the target application. These systems have been successfully deployed on popular devices from different manufacturers. The evaluation results show that they are effective and do not introduce noticeable system overhead.

2. REFERENCE HIJACKING

2.1 Approach Overview

An Android application needs to reference the underlying system components to perform its logic by invoking their provided interface methods. In general, the implementations of the underlying system components are stored in system library files, including *system class libraries* and *native libraries*. The execution environment of an application determines the location from which it seeks and loads the system library files. In Android, all application processes are forked from a special process called *Zygote*. It sets up an initialized execution environment for its child processes, in which the system library files will be loaded from designated system folders.

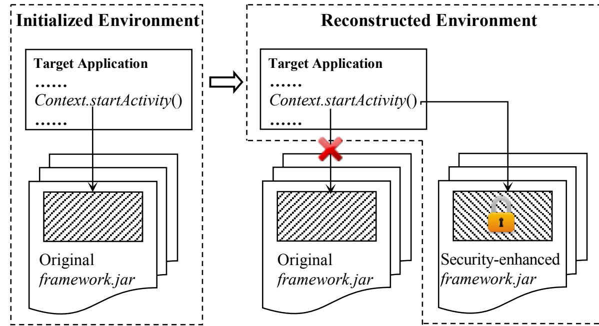

Figure 1. Reference Hijacking.

The goal of reference hijacking is to take control over the reference of an application to the underlying system libraries, so that it can be redirected to the security-enhanced alternatives. As illustrated in Fig. 1, by default, the target application runs in the initialized execution environment, in which the target application references to the original system libraries. For example, in order to start an Activity, an application invokes *Context.startActivity*() provided by the system class library *framework.jar* loaded from the */system/framework/* folder. With reference hijacking, we can reconstruct a new execution environment for the target application, in which it will load *framework.jar* from a configurable location rather than the original folder. We can implement a security-enhanced *framework.jar* to introduce extra security features (e.g., a security patch, an access control checking, or a taint tracking functionality) into *Context.startActivity*(), and let the target application load the security-enhanced library file rather than the original one. In this way, when the target application invokes *Context.startActivity*(), the introduced security feature will take effect simultaneously.

Reference hijacking is mainly achieved by a specially designed environment reset procedure. By modifying the startup process of an application, some additional operations are introduced to drive the application to restart with new environment variables (e.g., the library path). After restarting, the application will be executed in a new execution environment. To leverage reference hijacking, the target application should be adjusted. In particular, the application entry point will be rewritten with the code that invokes the environment reset procedure.

We generate the enhanced system libraries by either compiling the source code or directly rewriting the library files, depending on whether the target system libraries are open-source or not. The detailed steps for different security enhancement approaches are shown in Section 3 ~ 5 respectively. The generated system library files coexist with the original ones on the same device. They are placed in a specific folder that is readable but not writable for normal application processes. As such, the enhanced libraries can be securely shared by all applications of interest.

2.2 Environment Reset Procedure

In Android, the startup process of an application is under the control of the Activity Manager Service (AMS). AMS will create a new process for the target application by forking from Zygote when the first component of the application is activated. Besides, every application has an entry class, which is instantiated before any component of the application is launched. The default entry class is *Application*. An application can also declare a customized entry point class by inheriting the default Application class.

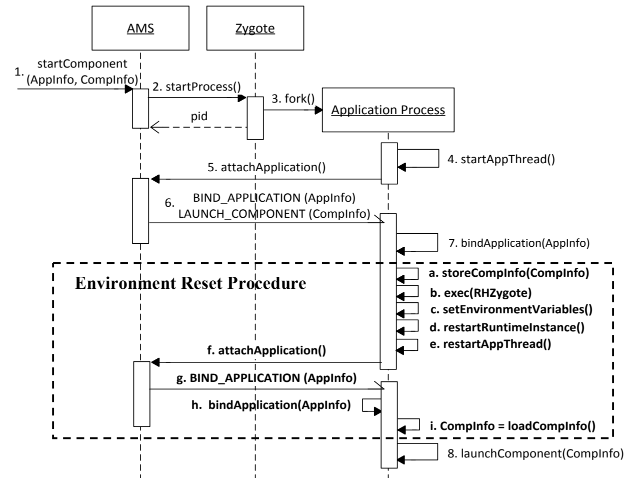

Figure 2. Sequence diagram of the application startup process. Additional steps introduced by the environment reset procedure are outlined with a dashed rectangle.

Fig. 2 depicts the startup process of an application. First, a *startComponent* request is sent to AMS with the information about the target application and the target component (Step 1). AMS then creates a new process for the target application by sending a *startProcess* request to Zygote (Step 2), making it fork a child process (Step 3). The forked process inherits the execution environment from Zygote and starts an application thread (Step 4), which will send an *attachApplication* request to AMS (Step 5). In response, the information about the target application and the target component is sent to the forked process via two synchronous messages (Step 6), which will be handled sequentially. Finally, the target application is started with its Application class being instantiated (Step 7), and the target component is launched (Step 8). In practice, manufactures often modify some aspects of Android. However, as a fundamental feature of Android, the application startup process always remains unchanged.

The environment reset procedure is invoked between Step 7 and Step 8. It is accomplished by a customized Application class *RHApplication* and an executable file *RHZygote*. RHApplication is used to store/resume necessary information and reset the current program state of the application process. RHZygote mimics the functionality of Zygote to construct a new execution environment. After Step 7 is finished, the RHApplication class is instantiated to perform Step *a* and Step *b*. Step *a*: store the target component information. The target component information will be discarded when the execution environment is reset. Hence, we store it in a temporary file. Step *b*: execute RHZygote. It is done by making a native *exec*() call to completely replace the current process with the RHZygote program. As such, the current program state of the target application is reset. Particularly, the reference of the target application to the original system libraries is cut off. The RHZygote program performs Step *c* to Step *e* to prepare a new execution environment. Step *c*: set environment variables. Two environment variables are set. One is `BOOTCLASSPATH` that specifies the security-enhanced system class libraries as default class paths. The other is `LD_LIBRARY_PATH` that specifies the location of the security-enhanced native libraries as the first search path of shared libraries. Step *d*: start a new runtime instance. The new runtime instance will load system class libraries and native libraries from the paths specified by the aforementioned environment variables.

Table 1. Evaluated devices.

Vendor	Device	Type	Version	Runtime	Delay
Motorola	Defy	MB525	2.2.2	Dalvik	1.10 s
HTC	Wildfire S	A510e	2.3.3	Dalvik	0.67 s
HTC	One S	T528w	4.0.4	Dalvik	1.23 s
HTC	Butterfly	X920e	4.1.1	Dalvik	0.57 s
Samsung	Galaxy S4	GT-I9500	4.2.2	Dalvik	0.65 s
Samsung	Galaxy Note3	SM-N9006	4.3	Dalvik	0.87 s
Samsung	Galaxy S5	SM-G900F	4.4.2	Dalvik	0.36 s
LG	Nexus 5	D820	4.4.2	ART	0.74 s
Motorola	Moto G	XT1079	5.0.2	ART	0.98 s
HUAWEI	Mate S	CRR-UL00	5.1.1	ART	1.08 s

Step *e*: restart an application thread. The restarted application thread will interact with AMS to reload the target application and instantiate the RHApplication class for a second time (Step *f* to Step *h*). At this time, RHApplication performs Step *i* to obtain the target component information from the temporary file. Finally, the target component will be launched for execution at Step 8.

2.3 Adjusting the Target Application

To make the target application invoke the environment reset procedure, we should adjust it to customize the RHApplication class as its entry class. The adjustment is automatically completed by a script tool written by us.

For the applications that do not declare a customized entry class, the RHApplication class is introduced into them by leveraging *Apktool* [3], an open-source repackaging tool. The bytecode of the target application is disassembled into the SMALI [14] intermediate language files. The RHApplication class is also converted to a SMALI code snippet and added to the set of the intermediate files. In addition, the `<application>` tag of the manifest file is altered to specify RHApplication as the entry class. After repackaging all the related files, a new package of the target application will be ready.

Some target applications may have already declared their own entry class. In this case, we will trace the inheritance of this class until finding the root base class, which directly inherits from the Application class. We modify the original class hierarchy, making the root base class inherit from the RHApplication class. In this way, RHApplication will always get executed before any of the application's own classes.

2.4 Deploying Security-Enhanced Libraries

Two issues need to be addressed for deploying security-enhanced libraries to devices. First, if we prepare an individual set of enhanced libraries for each target application, the space overhead would be extremely large. Second, if the enhanced libraries can be corrupted by malicious applications, the security enhancements relying on the libraries would be tricked or even cannot work.

To solve these issues, we package the security-enhanced libraries into an assistant application as its asset files. When the assistant application is installed on the device, these libraries will be released to a private folder of the application, which is set readable but non-writable for other applications. As such, these libraries are securely shared by all applications of interest.

2.5 Evaluation

We evaluate reference hijacking with some prevalent devices as detailed in Table 1. The evaluation involves almost all mainstream Android versions from 2.x to 5.x, and covers both the Dalvik and ART runtime environments. For each device, we use 10 applications of different categories to check whether they still work with reference hijacking. The evaluation result shows that reference hijacking can work well on all of these evaluated devices and does not affect normal functionalities of the target applications.

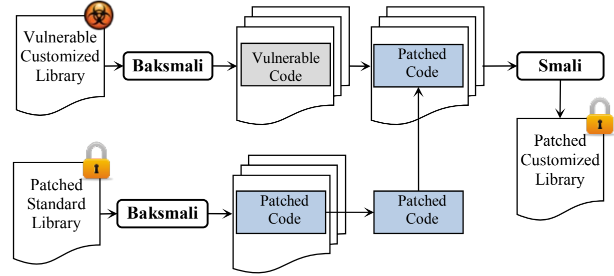

Figure 3. Rewriting system class library.

We also evaluate the performance overhead of reference hijacking. For each device, we launch 10 different applications each for 100 times before and after adopting reference hijacking respectively, and use the mean value of launch delay as the metric of performance. The evaluation results show that the application launch delay is less than 1.5 seconds. Note that the launch delay is primarily determined by the number of the pre-loaded classes (i.e., the classes that need to be loaded during the construction of the execution environment). The more pre-loaded classes there are in a device build, the more launch delays there will be. Nevertheless, since the launch delay only occurs when the first component of the target application is activated, the overhead is acceptable.

3. PatchMan

Unlike vulnerabilities of an application itself, the vulnerabilities of underlying system libraries can affect all applications using the vulnerable system library. Take the Heartbleed vulnerability [24] as an example. It is caused by missing necessary validation for untrusted data in the *OpenSSL* library. All OpenSSL protocol client applications executed on a vulnerable device will be affected by Heartbleed. We develop a prototype system, called PatchMan, to provide patched system libraries for the applications of interest without flashing or rooting devices. By leveraging reference hijacking, the target application can run on the top of the patched system libraries even on a vulnerable device.

3.1 Design and Implementation

In practice, for a specific vulnerability, we can get the patch from various channels, such as change logs of the Android source code tree [2], public vulnerability databases (e.g., CVE [5], OSVDB [8]), or security research organizations.

In most cases, we can fix a vulnerable library at source code level. We can prepare a compilation environment with the same configuration as the target device and integrate the patch code into the original source code of the vulnerable library. After compiling the fixed source code, a patched library will be ready for the device.

In some cases, manufactures may provide customized system libraries based on the standard Android implementation. The source code of the customized libraries is not always available. To fix the potential vulnerabilities in a close-source library, we need to directly rewrite the library file. Take patching a customized system class library as an example. We leverage two reverse engineering tools, *Baksmali* and *Smali* [14], to rewrite the Dalvik bytecode file of the library. As illustrated in Fig. 3, Baksmali is

Table 2. Evaluation result of PatchMan.

Vulnerability	Affected Version	Evaluated Device	Device Version	Block Attacks
CVE-2012-6636	< 4.2	HTC One S	4.0.4	✓
CVE-2014-0160 (Heartbleed)	4.1.1	HTC Butterfly	4.1.1	✓
Google Bug 13678484 (FakeID)	≤ 4.4	Samsung Galaxy S4	4.2.2	✓
CVE-2015-1287	4.4 ~ 5.1	LG Nexus 5	4.4.2	✓
CVE-2015-1210	4.4 ~ 5.1	Motorola MOTO G	5.0.2	✓

used to disassemble the vulnerable class library into a series of SMALI intermediate language files. Then, the vulnerable code is replaced with the patched code in the form of the intermediate language, which can be extracted from a patched standard class library. Finally, Smali is used to assemble the patched intermediate files to a new Dalvik bytecode file, generating an invulnerable customized library. In the similar way, we can leverage the existing binary rewriting technique [22] to patch a vulnerable close-source native library.

3.2 Evaluation of PatchMan

We choose five representative vulnerabilities to evaluate the effectiveness of PatchMan. The vulnerabilities are patched on some widely used vulnerable devices. We choose *LinkedIn* [6] as the target application to be protected. LinkedIn is a popular social application with millions of downloads. It maintains the private user profile and is granted with some dangerous permissions (e.g., `READ_CONTACTS` for reading contacts). If it is compromised by adversaries, the users will face serious security issues.

The evaluation is performed as follows. First, we exploit the vulnerability to attack LinkedIn installed on the vulnerable device. Then, we use PatchMan to patch the vulnerability in the system libraries and deploy the patched libraries on the device. Finally, we relaunch the attack to examine whether the patched libraries can successfully block the attacks. As shown in Table 2, the patched libraries can be successfully deployed on the tested devices and effectively block the attacks exploiting the vulnerabilities. In addition, it should be noted that these patches do not affect normal functionalities of LinkedIn. The protected LinkedIn can work well, and the performance overhead is also hardly noticeable.

3.3 Case Study

We choose the most severe two of the above vulnerabilities, including FakeID [16], Heartbleed [24], as cases to illustrate how to fix them by using PatchMan.

3.3.1 Patching the FakeID Vulnerability

FakeID (Google Bug 13678484) is a widespread vulnerability in the Package Manager Service (PMS). It allows malicious application to impersonate specially recognized trusted applications, and cause a wide spectrum of consequences. At the worst, it can be exploited to inject arbitrary code into the target application for execution. All devices prior to Android 4.4 are affected by FakeID. The proof-of-concept (PoC) exploit is available in [11].

FakeID is caused by a non-verification flaw in the application certificate chain enforced by PMS. An attacker can lure the user to install a malicious application, which exploits FakeID to forge the

```
01 boolean containsPluginPermissionAndSignatures(…){
02   ……
03   Signature signatures[] = pkgInfo.signatures;
04   signatures = getTrustableSignatures(signatures);
05   for (Signature signature : signatures) {
06     if (SIGNATURE.equals(signature)) return true;
07   }
08   ……
09 }
10 Signature[] getTrustableSignatures(Signature[] sigs) {
11   ……
12   for (int i = 1; i < sigs.length; i++) {
13     try { certs[i-1].verify(certs[i].getPublicKey()); }
14     catch (Exception e) { break; }
15   }
16   ……
17 }
```

Figure 4. Patch for the FakeID vulnerability. The patch code is presented in the bold italic type.

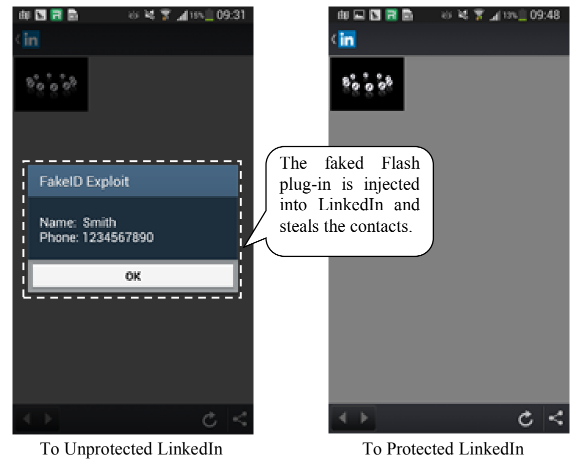

Figure 5. FakeID attacks to LinkedIn.

Adobe signature to impersonate as a Flash plug-in. When the user uses a WebView-based application (e.g., LinkedIn) to visit a Flash webpage, the malicious application will be treated as a Flash plug-in and be injected and executed with the permission of the victim application. This is a special kind of privilege escalation.

An early-bird patch of FakeID for vulnerable PMS is provided by Bluebox [9]. Because PMS is run as a system server process, we cannot patch it directly. Instead, we can create a new patch for the target application based on the idea of the patch provided by Bluebox. The new patch can also effectively fix FakeID in the target application (act as a PMS client) by introducing a validation of signatures in the system class library *framework.jar*. Specifically, we add the patch code in the method *containsPluginPermissionAndSignatures*() of the *PluginManager* class. As shown in Fig. 4, a filter method *getTrustableSignatures*() is added to revalidate the signatures provided by PMS (at line 4). The filter method will verify whether the child certificate is signed with the public key of the parent certificate (at line 13). Since manufactures (e.g., Samsung) often customize *framework.jar* to introduce non-standard features that is close-source, we choose to patch *framework.jar* by rewriting its Dalvik bytecode file as described in Section 3.1.

As a demonstration, Fig. 5 shows two screenshots of LinkedIn under attack. Before launching the attack, the user has been lured to install a faked Flash plug-in application, which does not pos-

```
01 if (1 + 2 + 16 > s->s3->rrec.length) return 0;
02 hbtype = *p++;
03 n2s(p, payload);
04 if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0;
05 pl = p;
```

Figure 6. Patch for the Heartbleed vulnerability.

sess the READ_CONTACTS permission. When the user visits a Flash webpage via LinkedIn on a vulnerable device, the faked Flash plug-in will be injected into LinkedIn and executed with the permissions of LinkedIn. It can get some private information from the contact list by leveraging the READ_CONTACTS permission of LinkedIn. Fortunately, after protecting LinkedIn with PatchMan, the malicious application will not be identified as a valid Flash plug-in. The attack can be effectively blocked. Besides, the patch does not affect the normal functionality of LinkedIn.

3.3.2 Patching the Heartbleed Vulnerability

Heartbleed (CVE-2014-0160) is a severe vulnerability in the popular OpenSSL cryptographic library. It allows attackers on the Internet to read up to 64 KB of the victim's memory. Devices equipped with Android 4.1.1 (e.g., HTC Butterfly) are affected by Heartbleed. The proof-of-concept exploit is available in [12].

The root cause of Heartbleed is lack of bounds checking for the length field of a heartbeat request. An attacker can exploit this flaw to attack LinkedIn users by luring them to visit a malicious HTTPS server that is controlled by the attacker. When visiting, the malicious server will send a crafted heartbeat request that contains small payload but claims a very large length. It will trick the OpenSSL library used by LinkedIn to respond with data from its memory space that most likely contains sensitive information, such as the user's profile.

The vulnerability exists in the native library *libssl.so*, which is directly generated from the OpenSSL project. We create the patch code for Heartbleed according to [10]. As shown in Fig. 6, two validations are added to the file *openssl/ssl/d1_both.c*. The first one (at line 1) is used to stop zero-length heartbeats; and the second one (at line 4) ensures the payload length is sufficiently long.

We generate a fixed *libssl.so* by integrating the patch code into the vulnerable source code and then recompiling the fixed code. With the help of reference hijacking, the protected LinkedIn application will use the fixed *libssl.so* to perform HTTPS communications to avoid suffering from Heartbleed attacks.

4. ControlMan

Android provides a powerful inter-component message passing system. Developers can conveniently leverage the existing resources and services provided by other applications. It promotes the development of rich applications, but also introduces new attack surfaces [34]. We develop a prototype system, called ControlMan, to provide a fine-grained access control over ICC of the target application. It allows end users to specify customized security policies for the target application, regulating: (1) which applications can serve specific ICC requests sent from the target application; and (2) which applications' ICC requests can be served by the target application.

4.1 Overview

Android defines four different types of application components, i.e., *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. Components interact with each other primarily via *Intent* messages, which declare a recipient and optionally contain data to be passed. Intents can be used explicitly or implicitly. An explicit Intent identifies the intended recipient by a fully-qualified name, while an implicit Intent leaves it up to the Android system to determine which application(s) should receive the Intent. In Android, AMS serves as a proxy for transferring the interactions among components.

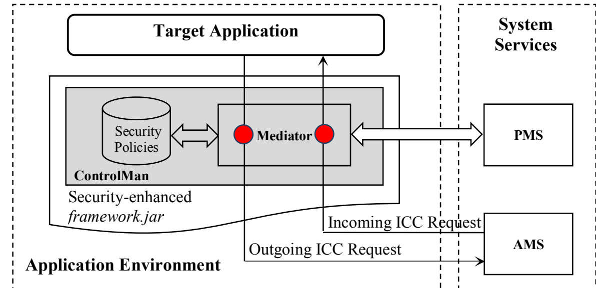

Figure 7. Architecture of ControlMan.

As shown in Fig. 7, ControlMan consists of two major parts: a security policy database storing the ICC control policies specified by users; and a mediator used to check whether an ICC request is allowed by the policies. Every outgoing and incoming ICC request should be validated by the mediator before delivering. The mediator is placed in the implementation of AMS client, which is located in the *framework.jar* class library.

By employing the method discussed in Section 3.1, we create an enhanced *framework.jar* equipped with the mediator. With reference hijacking, the applications of interest are forced to run with the enhanced *framework.jar* library to be protected.

4.2 Design and Implementation

4.2.1 Security Policies

The security policies of ControlMan specify which interactions between the target application and its opponent(s) are allowed. For outgoing ICC requests, the related security rule is a triple (*type*, *token*, *receivers*). The *type* item identifies the type of target components. The *token* item describes the content of the request. It can be an action string for an implicit Intent, a component name for an explicit Intent, or a URI for a Content Provider. The *receivers* item, as the name suggests, specifies the valid request receivers, which can be the signature, the version, or the permissions of an application. The security rule for incoming requests is a 2-tuple: (*component*, *senders*). The *component* item identifies the name of the target component of the request, and the *senders* item specifies the valid request senders. Since the type of incoming requests can be inferred from the component item, we do not need to include it in the policy rule.

4.2.2 Mediation of ICC Requests

Outgoing ICC requests can be divided into four types: *starting Activities*, *sending Broadcasts*, *binding Services*, and *accessing Content Providers*. Particularly, Activities are started with an Intent. We choose the *ActivityManagerProxy.startActivity*() method as the checkpoint, which will be called before a *startActivity* Intent is sent to AMS. The related mediation code is inserted into the method. When the method is invoked, the mediator will query PMS to retrieve all potential Activities that can be started by the current Intent. From them, the allowed ones are determined by matching with the related policy rules. If there is only one allowed

Table 3. Performance evaluation of ControlMan.

ICC Type		HTC One S Android 4.0.4		Motorola Moto G Android 5.0.2	
		Ori. (ms)	ControlMan (ms)	Ori. (ms)	ControlMan (ms)
Outgoing ICC Requests	Activity	71	81 (14.1%)	66	79 (19.7%)
	Broadcast	110	130 (18.2%)	46	54 (17.4%)
	Service	40	45 (12.5%)	33	37 (12.1%)
	ContentProvider	110	124 (12.7%)	75	86 (14.7%)
Incoming ICC Requests	Activity	190	226 (18.9%)	151	174 (15.2%)
	Broadcast	40	47 (17.5%)	33	40 (21.2%)
	Service	130	144 (10.8%)	127	141 (11.0%)
	ContentProvider	461	514 (11.5%)	368	411 (11.7%)

Activity, a new explicit Intent for starting this Activity will be constructed and sent to AMS. Otherwise, the user is prompted to choose one from the allowed Activities. In this case, an explicit Intent for the chosen Activity will be sent to AMS.

The mediation of other three types of outgoing requests is similar with that of starting Activities. For sending Broadcasts, the mediation code is placed in *ActivityManagerProxy.broadcastIntent*(). If there are more than one allowed Receivers, we will construct an explicit Intent for each of them and send these Intents to AMS one by one. The mediation code of binding Service requests is placed in *ActivityManagerProxy.bindService*(). If there are more than one allowed Services, the mediator also let the user to choose one from them. For a Content Provider, applications can perform the *query*, *insert*, or *delete* operations by invoking different methods. For each operation, we place the mediation code in the corresponding point where the operation request is about to be sent to the target Content Provider. For example, the mediation code for the query operation is placed in *ContentProviderProxy.query*().

The mediation for incoming ICC requests will be implemented in proper checkpoints similarly with done for the outgoing requests. However, there is a challenge in identifying the sender for checking an incoming Intent request of Activity and Broadcast. In Android, these two types of Intents do not contain the sender information. In practice, AMS will generate a log entry for an ICC request, recording the name of the sender and the name of the target component. We can infer the sender by reading the log. In addition, we can also leverage the usage statistics feature introduced in Android 5.0 to infer the sender as far as possible. This new feature provides the last active time of each application. Because ICC requests are processed in sequence, the sender of the latest received request is most likely to be the one whose last active time is just previous to that of the target application. We argue that it is trustable to collect the sender information from the system-level sources (e.g., the log and usage statistics), as long as the whole system is not compromised by malicious applications.

4.3 Evaluation of ControlMan

We have successfully deployed ControlMan on two prevalent devices: HTC One S (Android 4.0.4, Dalvik) and Motorola MOTO G (Android 5.0.2, ART). We choose these two devices to demonstrate that ControlMan can work in both the Dalvik and ART runtime environments. To precisely measure the performance overhead of ControlMan, we develop an experimental application to issue/serve all types of ICC requests. Besides, we define 40 security rules (5 for each ICC type respectively) to specify the valid ICC opponents for the application. For each type of ICC operation, we record its execution time before and after deploying ControlMan for 100 times and get the mean value as the metric. As shown in Table 3, the performance overhead of ControlMan is about 10% to 20%. We believe it is acceptable, since it only occurs in ICCs and does not interfere other operations.

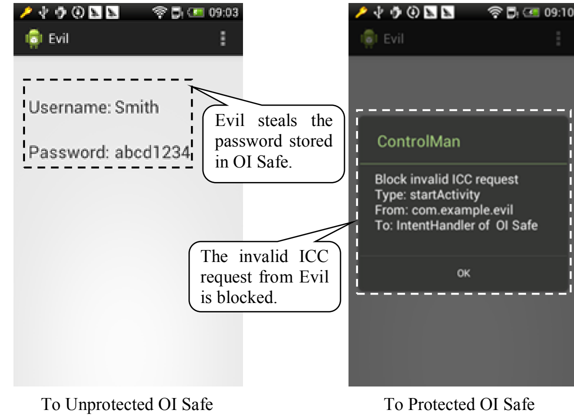

Figure 8. ICC attacks to OI Safe.

We also use real-world applications in the *OpenIntent* project [7] to evaluate ControlMan. For these applications, we define related security rules to protect them from potential ICC attacks. The experiment results show that ControlMan can effective block all ICC requests disallowed by the rule settings. From the applications, we choose *OI Safe* to demonstrate the effectiveness of ControlMan. OI Safe is a password manager, which is used to store user's passwords for other applications. In practice, OI Safe only allows applications possessing the `ACCESS_INTENT` permission to access its data via issuing an *IntentHandler* Intent. However, the user may install a malicious application with the `ACCESS_INTENT` permission by accident. To prevent privacy leakage as far as possible, we define security rules to further restrict OI Safe to only serve the *IntentHandler* request from the application with certain signatures (e.g., the signature of OpenIntent). As shown in Fig. 8, for the unprotected OI Safe, the password information stored in it can be stolen by an experimental malicious application *Evil*. By contrast, the malicious ICC request sent from Evil will be successfully blocked when protected by ControlMan.

5. TaintMan

For helping analysts to more effectively capture hidden malicious behaviors, we develop a prototype system, called TaintMan, to analyze the sensitive information flow on the off-the-shelf devices. TaintMan instruments both the target application and the underlying system libraries with taint policy code, and leverages reference hijacking to make the target application adopt the instrumented libraries. As such, it can provide an effective and efficient *whole-application-wide*, *instruction-level* DTA analysis for the applications of interest.

5.1 Overview

In TaintMan, the taint policy code tracking information flow is statically instrumented into both the target application and the underlying system class libraries. The rest of the system is not affected. This makes TaintMan can support both the Dalvik and ART runtime environments.

We implement a modified Baksmali tool to automatically instrument the target files. Baksmali is modified to generate additional taint policy code for each bytecode instruction when disassem-

Table 4. Taint propagation logic for the *binary-op* instruction. $\tau(v)$ stands for the taint tag of variable *v*.

Instruction	Coarse Propagation Logic	Condition	Refined Propagation Logic
binary-op v_A, v_B, v_C	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	$A \neq B \neq C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$
		$A=B$ && $A \neq C$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_C)$
		$A=C$ && $A \neq B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$
		$B=C$ && $A \neq B$	$\tau(v_A) \leftarrow \tau(v_B)$
		$A=B=C$	N/A

bling the target application or system class library. After disassembling, we will get the intermediate language files instrumented with taint policy code. They will be reassembled to a new application package or system library file by using Smali.

An application can obtain sensitive information (e.g., device identifier, phone number, location, SMS messages, contacts, etc.) by calling the corresponding framework APIs. TaintMan hooks these APIs (taint sources) to set a taint tag for the returned sensitive data. The propagation of the data is monitored by taint policy code during the execution of the target application. When a pre-defined sink point (e.g., network interface) is reached, the taint tags of the data will be checked to detect potentially malicious operation.

5.2 Design and Implementation

5.2.1 *Taint Tag Storage*

TaintMan provides a 32-bit vector for each variable to encode taint tags. Instead of modifying the runtime data structure to allocate additional space for taint tags, TaintMan requests taint tag storage by declaring additional variables.

When a method is invoked, its local variables and parameters are stored in the registers allocated on an internal stack. In order to store their taint tags, we expand the stack frame of the method to twice as large as its original size by doubling the number of the method's requested registers. The expanded space is used to store taint tag for each local variable and parameter. For each class field, its taint tag storage is allocated by inserting a shadow integer field into the class. There is a caveat that Android runtime has a restriction that certain fields of certain classes (e.g., the *value* field of the *String* class) should be placed at a fixed offset of the class structure. To avoid violating the restriction, the shadow fields should be placed after all of the original fields. For arrays, we store only one taint tag per array to minimize storage overhead. In the Dalvik bytecode language, array is a built-in class. We cannot add an additional shadow filed to the array class. To this end, we maintain a hash map between array objects and taint tags.

5.2.2 *Taint Tracking*

TaintMan adopts the classic taint propagation logic: given an instruction, the taint value of its destination operand is set to be the lower bound of the taint values of its source operand(s). In TaintMan, the taint propagation logic for an instruction is refined to require less taint policy instructions than the original one. To illustrate, we take the binary operation *binary-op* v_A, v_B, v_C as an example. The coarse taint propagation logic for the instruction is "$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$", where $\tau(v)$ stands for the taint tag of variable *v*. This logic can be refined as shown in Table 4. In particular, when the *binary-op* instruction takes the same register as its operands (i.e., $A = B = C$), no taint propagation is needed.

Even if a method could propagate taints across the method scope, it will not always propagate taints on each execution instance. Indeed, only when a method actually imports taints from the out-

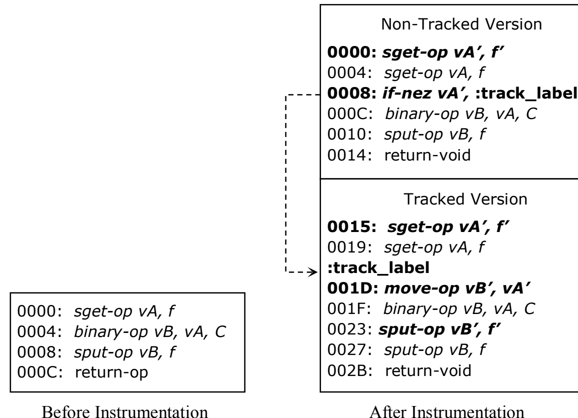

Figure 9. Bytecode of *Logger$Stream.endIndent*() for enforcing on-demand tracking. Instrumenting instructions are presented in the bold italic type, while the original instructions are presented in the normal type. *v*' and *f*' stand for the shadow register and shadow field respectively.

side, can it actually propagate taints. That means a method does not need to be tracked until it imports some taints from the outside. In practice, the instruction-level taint analysis often introduces high performance overhead due to a great deal of unnecessary taint tracking. To address this problem, TaintMan enforces *on-demand tracking* to reduce runtime cost of the taint analysis. Specifically, for each instrumented method, there are two versions of bytecode: a *non-tracked* version and a *tracked* version. When invoking a method, its non-tracked version is executed by default. The control will transfer to the tracked version when any taint is actually imported into the method.

The non-tracked version and the tracked version coexist in the instrumented method. The tracked version is placed next to the non-tracked version. In the non-tracked version, only data-importation instructions (DIIs) are monitored, which can import data from the outside of the method scope (e.g., the static field getter operation instruction *sget-op*). A conditional transfer instruction is added after each DII to transfer the control to the tracked version in the case that the imported data are tainted. The conditional transfer instruction uses a symbolic target address (i.e., label) to avoid complex offset computation. In the tracked version, all data operating instructions are monitored with taint policy code. For ease of transfer from the non-tracked version, we introduce a label after each DII. Note that the execution logic of the original method still remains unchanged, even if the control transfers from the non-tracked version to the tracked version.

Take the method *Logger$Stream.endIndent*(), selected from the system class libraries, as an example. This method is to get a value from a static field *f*, perform a binary operation, and put the result back to *f*. As shown in Fig. 9, the original method bytecode contains *sget-op*, *binary-op*, and *sput-op* instructions. After instrumentation, there are two versions of method bytecode. In the non-tracked version, only the *sget-op* instruction is monitored by a conditional control transfer instruction (at offset 0x0008). In the tracked version, all *sget-op*, *binary-op*, and *sput-op* instructions are monitored with taint tracking code. Besides, a label (before offset 0x001D) is introduced after the *sget-op* instruction. At runtime, when the *sget-op* v_A, *f* instruction imports taints from the static field *f*, the control will be transferred from offset 0x0008 in the non-tracked version to offset 0x001D in the tracked version.

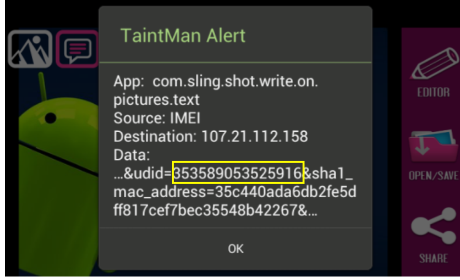

Figure 10. TaintMan can successfully detect privacy leakage.

Table 5. CaffeineMark benchmark results.

Test Items	HTC One S Android 4.0.4		Motorola Moto G Android 5.0.2	
	Ori.	**TaintMan**	**Ori.**	**TaintMan**
Sieve	1117	1048 (6.2%)	13409	12363 (7.8%)
Loop	893	686 (23.2%)	17506	13760 (21.4%)
Logic	1404	1366 (2.7%)	12700	12433 (2.1%)
String	893	596 (33.3%)	27802	18988 (31.7%)
Float	546	534 (2.2%)	7467	7236 (3.1%)
Method	675	595 (11.9%)	10818	9704 (10.3%)
Overall	***878***	***730 (16.9%)***	***13689***	***11745 (14.2%)***

5.3 Evaluation of TaintMan

We have successfully deployed TaintMan on two prevalent devices: HTC One S (Android 4.0.4, Dalvik), and Motorola Moto G (Android 5.0.2, ART). We choose these two devices to demonstrate that TaintMan can work in both the Dalvik and ART runtime environments. We mainly use HTC One S to evaluate the effectiveness of TaintMan with two test sets. The first set is 150 malware samples randomly selected from the Android Malware Genome Project dataset [1]. The second set is 100 popular real-world applications collected from the official market and some famous third-party markets. For confirming the detection results of TaintMan, we also deploy TaintDroid on an emulator. For each tested malware or application, we respectively execute it in both TaintMan and TaintDroid.

The malware detection result shows that TaintMan can successfully detect privacy-breaching behaviors from all these 150 tested samples. As a comparison, 147 of them are also proved by TaintDroid. The rest three are all from the *DroidKungFu3* malware family. The privacy-breaching behaviors of them are not triggered when they run in an emulator. Fortunately, when they are analyzed by TaintMan deployed in a real Android smartphone, their malicious behaviors are detected. Due to space limitation, we list the detailed detection result in our website[1].

Among all the 100 real-world applications, 51 are found to leak at least one kind of private information, of which 47 are detected by both TaintMan and TaintDroid. The rest four applications are detected only by TaintMan and cannot be executed in the TaintDroid environment. We manually analyzed these four applications and found they leak private information indeed. Take one of them, *Write on Pictures*, as an example. As shown in Fig. 10, we can see that the application actually leaks the IMEI number of the victim's device to a remote server. The potentially malicious behavior is successfully detected by TaintMan. Due to space limitation, we list the detailed detection result in our website[2].

From the above experiments, we can see that TaintMan has a comparative capability with TaintDroid in detecting privacy-breaching in malware samples and real-world applications.

We use CaffeineMark [4] to evaluate the performance overhead of TaintMan. CaffeineMark is a famous benchmark widely used by many Android-based security studies [38, 54]. It uses a series of tests to measure the performance of Java programs, and represents it as scores. These scores roughly correlate with the number of instructions executed per second. The evaluation result is shown in Table 5. We can see that the overall performance overhead is less than 20%. It is totally acceptable for the analysis purpose.

[1] http://iser.ruc.edu.cn/ReferenceHijacking/MalwareDetection.pdf.
[2] http://iser.ruc.edu.cn/ReferenceHijacking/ApplicationDetection.pdf.

6. DISCUSSION

The reference hijacking technique has been shown to be applicable to both the Dalvik and ART runtime environments in almost all mainstream Android versions (2.x to 5.x). In general, Google releases a new Android version in every several months [26]. At the time of writing this paper, Google releases Android 6.0 [17]. We are now working to make the reference hijacking technique applicable to it. The application launching process of Android 6.0 is similar as that of the previous versions. We believe the reference hijacking technique can also be applicable to Android 6.0 with minor modification. There is a risk that the future Android versions may make a great change in the application launching process. This may bring difficulties for adopting the reference hijacking technique in a certain future version. However, even at the worst case, this study is still of great value for enhancing security for numerous devices equipped with present Android versions.

Currently, reference hijacking requires repackaging the target application to hook its startup process. Some applications may validate the integrity of their packages, which may bring some troubles for repackaging. Additional efforts need to be taken to bypass the validation by adjusting the validation logic. A very recent work, Boxify [30], could help us to put forward a better way to hook the startup of applications. Boxify leverages the isolated process feature of Android to make the target application run in a monitored sandbox. The reference hijacking technique can be introduced in the isolated process to construct a new sandbox environment for the monitored application, making it run on the top of security-enhanced underlying system libraries. We believe that the combination of reference hijacking and Boxify can extend the capability of each other, and will result in a better solution for securing Android applications. We have begun to research on how to leverage Boxify to further improve the practicability of the reference hijacking technique.

In addition to the enhancements provided by the three prototype systems presented in this paper, the reference hijacking technique can be leveraged to implement more security features. For example, TaintMan can be extended to enforce an on-the-fly protection to block the command injection attack. In addition, the technique can also be used to update the underlying system libraries for the applications of interest to introduce some features in the fashion, not limited to the security enhancements.

Reference hijacking is a two-edged sword. In addition to securing Android devices, it can also be leveraged for attack purposes. In the Android security model, the underlying system libraries are treated as a part of trusted computing base (TCB). It is assumed that they would work as expected design goal to provide support for applications and interact with the system. However, this assumption can be broken down by leveraging reference hijacking. A malicious

application can make itself run with some malformed underlying system libraries. This may introduce some new attack surfaces. We performed a preliminary study on leveraging reference hijacking for attack purposes, and had some interesting findings. For example, we surprisingly found that the implementation of the Dalvik virtual machine interpreter can also be replaced via reference hijacking. As a result, attackers can design a special interpreter for their malware, such that the seemingly benign code will perform malicious behaviors during execution. It can be treated as an advanced obfuscation technique, which allows the malware to bypass the existing static program analysis techniques. In fact, the security issue is caused by the inherent characteristic of Android (i.e., the startup process of applications), rather than introduced by the reference hijacking technique. In the future, we will research on how to mitigate the security risks.

7. RELATED WORK

Erlingsson proposed the concept of inline reference monitoring in his thesis [39]. The basic idea is to inline the security monitoring code into the target program. Some existing studies [35, 36, 42, 51] have employed IRM to enforce security enhancements. In general, they rewrite the target program to invoke security checks before performing some certain operations of interest, e.g. calling a security-sensitive API. However, it is difficult to enforce an in-depth security enhancement only by intercepting API calls, such as completely repairing a vulnerable system library or tracking the execution of an underlying function.

Xposed [15] provides a security module development framework. It hijacks the Zygote process by replacing its executable file (i.e., *app_process*) with a modified one. The modified Zygote loads the customized modules to every instance of the Dalvik virtual machine, allowing every method call to be hooked. Some security approaches [45, 46] have been designed base on Xposed. However, Xposed requires rooting devices to hijack Zygote, and only supports limited devices and Android versions [27].

Some existing studies concentrate on patching vulnerabilities located in the application [29, 53] or in the underlying system components [43, 50]. For the application-layer vulnerabilities, they can be directly patched by rewriting the bytecode of the vulnerable applications. In order to patch system components vulnerabilities, PatchDroid [43] hooks critical system processes (e.g., Zygote) to dynamically inject patch code into the target application process. However, the hooking of system processes requires root privilege. As a result, PatchDroid can only work in rooted devices. Similarly, the study in [50] also requires rooting devices to patch system libraries vulnerabilities. As a comparison, PatchMan can patch system libraries vulnerabilities for the applications of interest without rooting devices.

A number of studies have been proposed to enhance the Android security model to provide fine-grained and flexible access control on sensitive APIs [42, 51, 54], on application components [37, 44], or on whole system resources [32]. Some of them [42, 51] achieve their goals by rewriting applications to hook critical function calls. For example, Dr. Android and Mr. Hide [42] rewrite the bytecode of the target application to encapsulate sensitive APIs with monitoring code. However, these studies are limited to enforcing the function-call-level protection policy and are insufficient in enforcing in-depth security enhancements. Other studies [32, 37, 44, 54] require flashing devices to embedded their implementation into the underlying system components. As demonstrated by ControlMan, with reference hijacking, we can implement most of the core functionalities of these studies for the applications of interest without flashing devices.

A recent work, Boxify [30], provides a novel privilege separation solution. The untrusted application is sandboxed and run in an isolated process. In the sandbox, IPC calls and system calls of the target application can be intercepted and mediated. Boxify does not require flashing or rooting devices, neither require modifying the target application. It is a great improvement in securing Android. However, it is still limited to enforcing the function-call-level protection policy. In this sense, Boxify and our study can perfectly complement each other to create a better solution. We leave the combination of them as our future work.

Many malware analysis techniques [28, 38, 40, 41, 47, 52] have been proposed. FlowDroid [28] is a representative static analysis approach on Android platform. It performs a precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for each Android application component. In this study, we mainly focus on the dynamic analysis. DTA is a mainstream malware dynamic analysis technique. Many analysis systems are proposed based on DTA. DroidScope [52] is a virtualization-based malware analysis platform. It is built on the top of a CPU emulator named QEMU [31] and cannot be deployed on real devices. TaintDroid [38], an extension to Android, provides real-time information flow tracking by modifying the Dalvik virtual machine of Android. AppFence [41] extends TaintDroid, allowing users to feed fake private information to applications and block network transmissions that contain private information. If users want to employ these two systems, they have to flash devices. AppCaulk [47] does not require modifying the underlying system. However, it employs a summary-based approach to analysis underlying system library methods, which is not precise enough. By contrast, TaintMan can perform instruction-level taint analysis for the target application and its underlying system libraries on an unmodified device.

8. CONCLUSION

In this paper, we propose a novel technique, called reference hijacking, to enforce in-depth security enhancements in the underlying system libraries of the target applications. Compared with the existing studies, it does not require any modification of the underlying system. Based on reference hijacking, three prototype systems, PatchMan, ControlMan, and TaintMan, are implemented to patch vulnerabilities, protect ICC communications, and perform DTA analysis for the target application respectively. By evaluating them on a number of off-the-shelf devices, the reference hijacking technique and these systems have shown to be applicable to both the Dalvik and ART environments and to almost all mainstream Android versions (2.x to 5.x). Overall, our work provides a new way to more practicably enhance the security of Android devices. We hope that this study can be combined with other efforts to get better solution and address more security problems.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. This research work was supported, in part, by National Natural Science Foundation of China (NSFC) under grants 61170240, 91418206 and 61472429, National Science and Technology Major Project of China under grant 2012ZX01039-004, NSF under awards 1409668, 1320326, and 0845870, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

10. REFERENCES

[1] Android Malware Genome Project. http://www.malgenomeproject.org.

[2] Android Source Code Tree. https://android.googlesource.com.

[3] Apktool. http://ibotpeaches.github.io/Apktool.

[4] CaffeineMark. http://www.benchmarkhq.ru/cm30.

[5] CVE. https://cve.mitre.org.

[6] LinkedIn. https://play.google.com/store/apps/details?id=com.linkedin.android.

[7] OpentIntent. https://code.google.com/p/openintents/downloads/list.

[8] OSVDB. http://osvdb.org.

[9] Patch for FakeID. https://android.googlesource.com/platform/libcore/+/android-cts-4.1_r4.

[10] Patch for Heartbleed. http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902.

[11] PoC for FakeID. https://github.com/retme7/FakeID_poc_by_retme_bug_13678484.

[12] PoC for Heartbleed. https://github.com/Lekensteyn/pacemaker.

[13] Proportion of devices running vulnerable versions of Android. http://androidvulnerabilities.org/graph.

[14] Smali and BakSmali. https://github.com/JesusFreke/smali.

[15] Xposed Framework. http://repo.xposed.info.

[16] Bluebox. Android Fake ID Vulnerability Lets Malware Impersonate Trusted Applications. https://bluebox.com/technical/android-fake-id-vulnerability.

[17] CNET. M is for Marshmallow: Google names its next Android update. http://www.cnet.com/news/m-is-for-marshmallow-google-announces-name-of-android-update.

[18] ExtremeTech. Google Throws Nearly a Billion Android Users Under the Bus, Refuses to Patch OS Vulnerability. http://www.extremetech.com/mobile/197346-google-throws-nearly-a-billion-android-users-under-the-bus-refuses-to-patch-os-vulnerability.

[19] Forbes. Report: 97% Of Mobile Malware Is On Android. http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe.

[20] Inquirer. Outdated Android Devices Are Exposing 400 Million Users to Security Threats. http://www.theinquirer.net/inquirer/feature/2235734/outdated-android-devices-are-exposing-400-million-users-to-security-threats.

[21] International Data Corporation. Smartphone OS Market Share, 2015 Q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[22] ISPRAS. Static ARM Binary Code Instrumentation. http://www.ispras.ru/en/technologies/static_arm_binary_code_instrumentation/.

[23] Neowin. HTC Officially Stops One S Updates, Barely a Year after Launch. http://www.neowin.net/news/htc-officially-stops-one-s-updates-barely-a-year-after-launch.

[24] OpenSSL. The Heartbleed Bug. http://heartbleed.com.

[25] TechTarget. Rooted Android Device Risks Include Network Access, Data Theft. http://searchmobilecomputing.techtarget.com/tip/Rooted-Android-device-risks-include-network-access-data-theft.

[26] Wikipedia. Android Version History. https://en.wikipedia.org/wiki/Android_version_history.

[27] Xda-Developers. List of Devices Able to Run Xposed on Lollipop. http://forum.xda-developers.com/xposed/list-devices-able-to-run-xposed-lollipop-t3030993.

[28] S. Arzt, S. Rasthofer, C. Fritz , E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI' 2014.

[29] T. Azim, I. Neamtiu, and L. M. Marvel. Towards Self-Healing Smartphone Software via Automated Patching. In *Proceedings of the 2014 International Conference on Automated Software Engineering*. ASE' 2014. ACM, 623-628.

[30] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. S. Rekowsky. Boxify: Full-Fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Security Symposium*. Security' 2015.

[31] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In Proceedings of the USENIX Annual Technical Conference. ATC' 2005. USENIX Association, 41-46.

[32] S. Bugiel, S Heuser, and A. R. Sadeghi. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22th USENIX Security Symposium*. Security' 2013. USENIX Association, 131-146.

[33] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24st USENIX Conference on Security Symposium*. Security' 2015, USENIX Association, 659-674.

[34] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems*. MobiSys' 2011. ACM, 239-252.

[35] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-Contract on the .NET Platform. Technical Report. https://lirias.kuleuven.be/bitstream/123456789/183893/1/Elsevier%2BInformation%2BSecurity%2BTechnical%2BReport.pdf.

[36] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. A Flexible Security Architecture to Support Third-Party Applications on Mobile Devices. In *Proceedings of the 2007 ACM workshop on Computer Security Architecture*. CASW' 2007. ACM, 19-28.

[37] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating

Systems. In *Proceedings of the 20th USENIX Security Symposium*. Security' 2011.

[38] W. Enck, P. Gilbert, B. G. Chun, L. P. 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*. OSDI' 2010. USENIX Association, 393-407.

[39] Ú. Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. *PhD Dissertation, Cornell University.* 2004. http://www.ru.is/starfsfolk/ulfar/thesis.pdf.

[40] C. Gibler, J. Crussell, J. Erickson, H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of 5th International Conference on Trust and Trustworthy Computing*. TRUST' 2012. Springer, 291-307.

[41] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall. "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. ACM, 639-652.

[42] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. D. Millstein. Dr. Android and Mr. Hide: Fine-Grained Security Policies on Unmodified Android. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM' 2012. ACM, 3-14.

[43] C. Mulliner, J. Oberheide, W. K. Robertson, and E. Kirda. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACSAC' 2013. ACM, 259-268.

[44] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*. ACSAC' 2009. ACM, 340-349.

[45] G. Paul and J. M. Irvine. Achieving Optional Android Permissions without Operating System Modifications. In *Proceedings of the 81st Vehicular Technology Conference*. VTC' 2015. IEEE, 1-5.

[46] L. Qiu, Z. Zhang, Z. Shen, and G. Sun. AppTrace: Dynamic Trace on Android Devices. In *Proceedings of the 2015 International Conference on Communications*. ICC' 2015.

[47] J. Schütte, D. Titze, and J. M. D. Fuentes. AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking into Android Apps. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom' 2014. IEEE, 370-379.

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of 31st IEEE Symposium on Security and Privacy*. SP' 2010. IEEE, 317-331.

[49] T. Vidas and N. Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIACCS' 2014. ACM, 447-458.

[50] L. Wu. Vulnerability Detection and Mitigation in Commodity Android Devices. *PhD Dissertation, North Carolina State University*. 2015. http://repository.lib.ncsu.edu/ir/bitstream/1840.16/10496/1/etd.pdf.

[51] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*. Security' 2012, USENIX Association, 539-552.

[52] L. K. 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*. Security' 2012. USENIX Association, 569-584.

[53] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*. NDSS' 2014.

[54] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*. TRUST' 2011. Springer, 93-107.

[55] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP' 2012. IEEE, 95-109.