TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices

Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang, *Member, IEEE*

Abstract—Dynamic taint analysis (DTA), as a mainstream information flow tracking technique, has been widely used in mobile security. On the Android platform, the existing DTA approaches are typically implemented by instrumenting the Dalvik virtual machine (DVM) interpreter or the Android emulator with taint enforcement code. The most prominent problem of the interpreter-based approaches is that they cannot work in the new Android RunTime (ART) environment introduced since the 5.0 release. For the emulator-based approaches, the most prominent problem is that they cannot be deployed on real devices. In addition, almost all the existing Android DTA approaches only concern the explicit information flow caused by data dependence, while completely ignore the impact of implicit information flow caused by control dependence. These problems limit their adoption in the latest Android system and make them ineffective in detecting the state-of-the-art malware whose privacy-breaching behaviors are inactivated in the analyzed environment (e.g., the emulator) or conducted via implicit information flow. In this paper, we present TaintMan, an ART-compatible DTA framework that can be deployed on unmodified and non-rooted Android devices. In TaintMan, the taint enforcement code is statically instrumented into both the target application and the system class libraries to track data flow and common control flow. A specially designed execution environment reconstruction technique, named reference hijacking, is proposed to force the target application to reference the instrumented system class libraries. By enforcing on-demand instrumentation and on-demand tracking, the performance overhead is significantly reduced. We have developed TaintMan and deployed it on two popular stock smartphones (HTC One S equipped with Android-4.0 and Motorola MOTO G equipped with Android-5.0). The evaluation with malware samples and real-world applications shows that TaintMan can effectively detect privacy leakage behaviors with an acceptable performance overhead.

Index Terms—Dynamic taint analysis, information flow control, privacy leakage, static instrumentation, android, dalvik, DVM, ART

1 Introduction

Android has become the most widely used mobile operating system. It dominated the global smartphone market with an 87 percent share in 2016 and continues to grow steadily [1]. Meanwhile, such popularity of Android also makes it more attractive to adversaries. A security report from F-Secure highlights that Android accounts for 97 percent of all mobile malware [2]. Even in the official application market (i.e., Google Play), there still exist a considerable number of malware [3].

Among all the malicious activities of Android malware, the most common one is stealing private information. As shown in the work of Zhou et al. [4], 744 (59 percent) of the total 1,260 Android malware samples collected from several markets have been found to actively collect various private information on the infected phones. Since mobile users are increasingly relying on smartphones to store and handle their personal data, the privacy-breaching malware will pose a more significant threat to user privacy in the future.

In order to detect privacy leakage, there is a need for an in-depth inspection of how private information is actually used by an application. Dynamic taint analysis (DTA) [5], as a mainstream information flow control technique, is quite suitable for such a task. It can precisely monitor sensitive information flow during application execution to examine whether private data is transmitted out of the device.

On the Android platform, with the evolution of system runtime and the development of malware technique, three rational but challenging demands are proposed on the design and implementation of a DTA framework. First, it requires the DTA framework to be workable in the new Android RunTime (ART) environment introduced since the 5.0 release. Second, it requires the DTA framework to be deployable on real devices for capturing evasive behaviors that are inactivated in the analyzed environment (e.g., the emulator). Third, it requires the DTA framework to be feasible to mitigate the security threat caused by common implicit information flow.

There are some existing DTA approaches for Android, of which TaintDroid [6] and DroidScope [7] are the two representative ones. TaintDroid provides a realtime system-wide information-flow tracking by instrumenting the Dalvik virtual machine (DVM) interpreter. It is the core of many malware detection systems [8], [9]. DroidScope provides

- *W. You, B. Liang, W. Shi and P. Wang are with the School of Information, Renmin University of China, Beijing, China and also with Key Laboratory of DEKE (Renmin University of China), MOE, China. E-mail: {youwei, liangb, wenchang, pengwang}@ruc.edu.cn.*
- *X. Zhang is with Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: xyzhang@cs.purdue.edu.*

Manuscript received 24 May 2016; revised 27 June 2017; accepted 4 Aug. 2017. Date of publication 15 Aug. 2017; date of current version 16 Jan. 2020. (Corresponding author: Bin Liang.)
Digital Object Identifier no. 10.1109/TDSC.2017.2740169

virtualization-based malware analysis support by instrumenting the Android emulator. With DroidScope, analysts can perform information flow analysis of the whole Android system running in the emulator.

These existing DTA approaches have proven to be valuable in analyzing privacy leakage behaviors in the past few years. Unfortunately, they do not fully satisfy the aforementioned three real-world demands. Most prominent problem of the TaintDroid-like interpreter-based approaches is that they cannot work in the ART runtime environment. In ART, the DVM interpreter is replaced with an on-device compiler suite. Consequently, the interpreter-based approaches lack their instrumentation target and hence are not applicable anymore. For those DroidScope-like emulator-based approaches, the most prominent problem is that they cannot be deployed on real devices. Nowadays, sophisticated malware samples have begun to employ anti-analysis technique to evade detection. Especially, they will inactivate their malicious logic if they perceive that they are not executed on the real device [10]. Consequently, the emulator-based approaches are ineffective in capturing evasive behaviors of the state-of-the-art malware. In addition, almost all the existing Android DTA approaches only concern explicit information flow caused by data dependence, while completely ignore the impact of implicit information flow (IIF) caused by control dependence. Our prior work [11] has demonstrated the effectiveness and efficiency of IIF in transmitting sensitive data. Real-world malware samples are found to leverage IIF. It is not only a theoretical threat but a reality.

Based on the above discussion, we argue that it is necessary to design and implement a DTA framework for the ART runtime environment on real smartphone devices to track data flow and common control flow. In this paper, we present TaintMan, an ART-compatible DTA framework that can be conveniently deployed on unmodified and non-rooted stock Android devices. To mitigate the threat of IIF, we develop a tracking algorithm based on our prior work [11] to track a special kind of control dependence called *strict control dependence*, which highly resembles the nature of data dependence and hence is most likely to be leveraged for attacks. Different from most existing DTA approaches, TaintMan is implemented via static instrumentation, rather than dynamic instrumentation. Given a suspicious application, analysts can use TaintMan to automatically instrument its bytecode file(s) with taint enforcement code that achieves information-flow tracking. The instrumented application is installed and run on the smartphone directly. During execution, taint tracking will be performed simultaneously as if it were a part of the application's functionality.

It should be noted that tainted data can be propagated via both application code and system class code. It is insufficient to only instrument application code for tracking taint propagation paths involving system class libraries. In TaintMan, to address the problem, both the target application and the system class libraries are instrumented with taint enforcement code. By instrumenting system class libraries, we can provide an instruction-level taint tracking completely covering the underlying system classes. It is more precise than the approaches relying on method summaries to model the behaviors of the underlying system classes [12], [13].

However, making the target application adopt the instrumented libraries is not a trivial task. In Android, system class libraries are placed in a specific system folder that is only writable for the *root* user. Without rooting the device, it is impossible to rewrite or replace the original system class libraries with their instrumented counterparts. To this end, we propose *reference hijacking*, a novel technique to reconstruct a new execution environment for the target application, where the system class libraries can be loaded from a configurable location instead of the default folder. With this technique, the reference of the target application to the system class libraries can be redirected to their instrumented counterparts. Eventually, the information flow involving the underlying system classes can also be effectively tracked.

For a DTA framework to be practical, its performance overhead should be acceptable. The most existing DTA approaches always instrument and track all instructions without discrimination, hence suffer from an unnecessarily high performance overhead. In TaintMan, we enforce on-demand instrumentation and on-demand tracking to optimize the performance. First, static analysis is conducted to identify the methods that can propagate taints across method scope. Second, two versions of bytecode are prepared for each identified method: a *non-tracked* version and a *tracked* version. The version to be executed is determined at runtime by observing whether the method actually imports taint from the scope outside the method. As such, the instrumentation and tracking only occur when necessary. In addition, we store the taint tags in an efficient way so that they can be conveniently accessed. We also refine the taint propagation logic for each kind of instruction to allow taint tracking to be implemented with as little code as possible. These elaborate designs dramatically reduce the performance overhead.

We have developed TaintMan and deployed it on two popular stock smartphones: HTC One S equipped with Android-4.0 and Motorola MOTO G equipped with Android-5.0. We evaluate TaintMan with three sets of application samples: 150 malware samples selected from the Android Malware Genome Project [14], 100 popular applications collected from multiple markets [15], [16], [17], [18], and 9 proof-of-concepts and 2 real-world malware samples leveraging IIF [11]. The evaluation results show that TaintMan can effectively detect privacy leakage behaviors. In addition, the performance and storage overhead of TaintMan are acceptable for analysis purposes. The evaluation with a standard benchmark shows that TaintMan incurs 42.3 percent performance overhead for tracking both data dependence and strict control dependence without optimization and 28.9 percent with optimization. The evaluation with real-world applications shows that TaintMan has no noticeable interference on the interactive behaviors of applications. The size of the instrumented applications is about 23 percent larger than the original ones, and the size of the instrumented system class libraries are about three times as large as the original ones.

In summary, the main contributions of this paper are the following:

- We present TaintMan, an ART-compatible dynamic taint analysis framework that can be conveniently deployed on stock smartphones without flashing or rooting devices.

- We enhance TaintMan with a tracking algorithm to track a special kind of control dependence, which highly resembles the nature of data dependence and hence is most likely to be leveraged for attacks.
- We propose a novel execution environment re-construction technique to force the target application to reference the instrumented system class libraries. As a result, taint tracking can completely cover both the target application code and the system classes at the instruction level.
- We enforce on-demand instrumentation and on-demand tracking to avoid unnecessary taint analysis whenever possible. In addition, we design efficient taint tag storage and refine taint propagation logic to implement taint tracking with as little code as possible. These optimizations dramatically reduce the performance overhead.

2 Background

2.1 Android Application Model

Android applications are mainly developed in Java, and converted into the customized Dalvik bytecode langauge format to be stored in bytecode file(s). [1] In the DVM runtime environment (Android version 4.4 and below), the bytecode is interpreted by the Dalvik virtual machine at runtime, with hot traces and functions being just-in-time (JIT) compiled into native code for execution. In the ART runtime environment (Android version 5.0 and above), the bytecode is completely compiled into native code at install time and executed directly at runtime without any interpretation.

The lifecycle of an application process is managed by the Activity Manager Service (AMS). In particular, when starting an application, AMS will create a new process for it by forking from a special process, called *Zygote*. The Zygote process is created during the system boot-strap. It initializes an execution environment, which will be inherited by all forked application processes. Especially, the execution environment inherited from Zygote will load system classes from the default system class libraries.

Android provides a default *Application* class as the entry point of an application. Developers can specify customized Application class in the application's manifest file. When an application is about to start, its Application class is instantiated. The class initialization method of the Application class will be invoked before any component is activated. [2]

2.2 Dalvik Bytecode Language

Dalvik bytecode language is register based. All computations are performed via registers. Dalvik has six kinds of variables, including local variables, parameters (actual or formal), return values, exceptions, class fields (static or instance) and arrays. Values of local variables and parameters are stored in registers and can be directly manipulated. Return values are moved from the callee's registers to the caller's registers after invocation. Exceptions are passed from the registers at the exception site to the registers in the exception handler. Values of class fields and array elements are loaded from and stored to registers before and after use.

The instruction set of Dalvik bytecode language has a variable length. The length of an instruction is decided by both the number of its operands and the size of each operand. Every instruction has certain restrictions on the maximal index of its operand registers. In the case that an instruction has to manipulate a register whose index exceeds the index restriction, it is expected that the register content get moved from the original register to a lower-indexed register before operation, and moved from the lower-indexed result register to the higher-indexed register after operation.

3 Approach Overview

TaintMan has two major components: an instrumentation tool that runs on the desktop computer, and a reference hijacking tool that runs on the smartphone device. The instrumentation tool, named *Instrumentor*, is used for statically instrumenting both the target application and the system class libraries. It is implemented on top of Smali/Baksmali [19], an open-source assembler/disassembler for the Dalvik bytecode language. The reference hijacking tool is used for reconstructing a new execution environment for the target application, forcing it to reference the instrumented underlying libraries. It is accomplished by a customized Application class *RHApplication* and an executable program file *RHZygote*. RHApplication is used to store/resume necessary information and reset the current program state of the application process. RHZygote mimics the function of Zygote to construct a new execution environment.

Fig. 1a shows the instrumentation procedure of an application. First, a reverse engineering tool, Apktool [20], is employed to decompress the original application package. Second, by using the Instrumentor tool, the taint enforcement code is added to the original bytecode file(s). Third, the manifest file is modified to alter the entry class of the target application, making it point to the reference hijacking procedure. Finally, these modified files along with the remaining resource files are packed, generating the instrumented application. The generated application will be installed on the device as the substitution of the original counterpart.

Fig. 1b shows the instrumentation procedure of a system class library. First, the system class library file is exported from the device. Then, the Instrumentor tool is used to instrument the original system class library with the taint enforcement code, generating the instrumented system class library file. Finally, the instrumented system class library file is imported into the device and placed in a specific folder. This folder is set to be readable but non-writable for normal application processes. As a result, the instrumented system class library file can be securely shared by all applications for space saving. [3]

1. By default, every application has a single .dex file. If the application code grows beyond the limits of what is allowed in a single .dex file (e.g., number of classes, number of methods, etc.), the code will be split over multiple .dex files.

2. We treat the class initialization method (rather than the *onCreate*() method) of the *Application* class as the entry point of an application. It is always executed before the initialization of any component.

3. We package the instrumented 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.

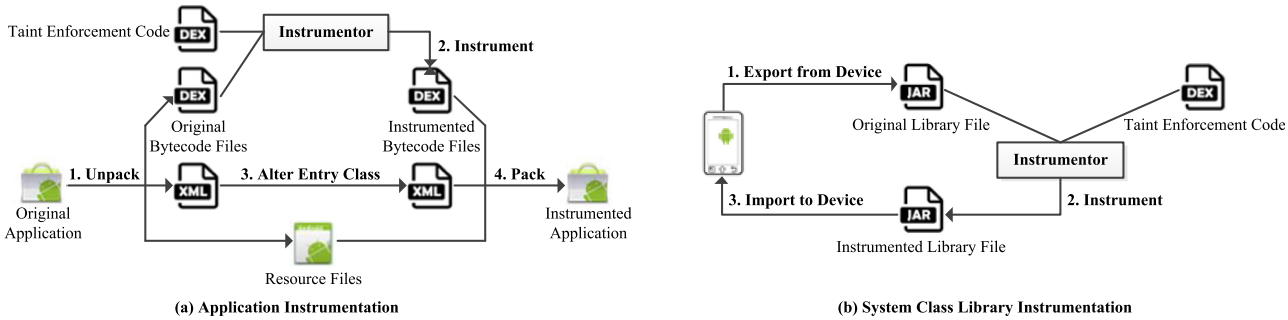


Fig. 1. Instrumentation of an application and a system class library. The instrumentation process is performed on the desktop computer.

The overall workflow of the Instrumentor tool is shown in Fig. 2. Given an application or a library, Instrumentor first performs static analysis to identify the taint-related methods, as well as computes auxiliary information, such as control flow graph (CFG), post dominator tree (PDT), and static single assignment (SSA). Then, the identified methods are instrumented with the traditional data flow tracking code (detailed in Section 4.2) and strict control dependence tracking code generated under the help of the auxiliary information (detailed in Section 4.3).

When the instrumented application runs on the device, the customized RHApplication class is instantiated and its class initialization method is invoked to execute the RHZygote program. As a result, reference hijacking will be adopted to reconstruct a new execution environment for the target application, in which the instrumented system class libraries are referenced instead of the original ones. During the application execution, the taint enforcement code is executed to enforce the taint tracking functionality. Specifically, when the application accesses private information, taint tags are attached to the variables storing the private data. These taint tags are propagated to other variables whose values are transitively derived from the tainted variables. When the tainted data are about to go out of the application scope, a dialog box is shown to inform the analysts about the source, destination and content of the tainted data.

4 Design and Implementation

TaintMan provides an effective and efficient application-wide, instruction-level, variable-granularity dynamic taint analysis for Android applications. This section illustrates the detailed design and implementation issues of TaintMan, including: (1) how to store taint tags; (2) how to implement taint tracking for data dependence and strict control dependence; (3) how to enforce on-demand instrumentation and on-demand tracking optimization; and (4) how to reconstruct the execution environment for referencing to the instrumented underlying libraries.

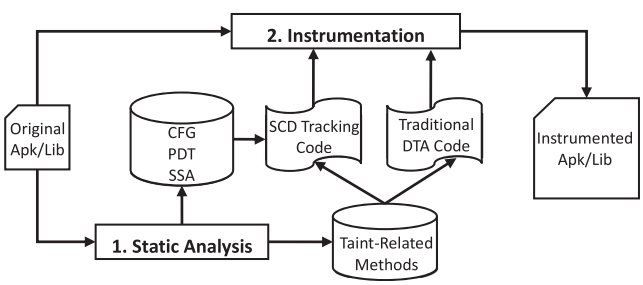


Fig. 2. Workflow of the *Instrumentor* tool.

4.1 Taint Tag Storage

TaintMan provides a 32-bit vector for each Dalvik variable to encode taint tag, allowing at most 32 different taint markings. *Instead of directly modifying the internal runtime data structure to allocate extra space for taint tags, TaintMan requests taint storage by declaring extra variables in the bytecode.* Great efforts need to be taken to ensure that the taint variables can be accessed directly within the application context, and to ensure that the allocation of taint variables would not violate the semantic restrictions of the Dalvik bytecode language.

4.1.1 Local Variables and Formal Parameters

In Dalvik, both local variables and formal parameters are stored in the registers allocated on an internal stack. When a method is invoked, a new stack frame is created for its registers. Particularly, a method's k formal parameters are always located in the last k registers of the stack frame. In order to store the taint tags for each local variable and formal parameter in the stack frame, we should expand the stack frame to twice as large as its original size by doubling the number of the method's requested registers. Besides, an extra register is allocated, which will be used as a temporary register during taint propagation. Note that the expansion of the stack frame only impacts the number of local registers and has no impact on the number of parameters. The taint tags of parameters are stored in the expanded stack frame as normal local registers. Consequently, the method prototype remains unchanged.

Fig. 3 illustrates how taint tags are stored for local variables and formal parameters. Given a method requesting two registers for a single local variable and a single formal parameter, the original stack frame of the method is shown in Fig. 3a. After increasing the number of the requested registers, the expanded stack frame stores five registers, as shown in Fig. 3b. Formal parameter $para0$ stored in register

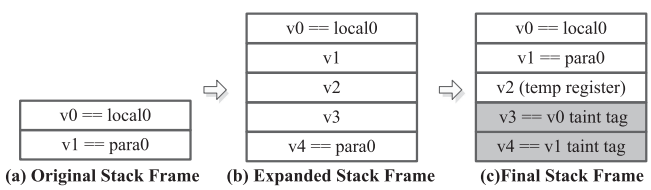


Fig. 3. Taint tag storage for local variables and formal parameters. Shadow registers are presented as grayed boxes.

$v1$ of the original stack frame will be stored in register $v4$ of the expanded stack frame. This may lead to application crash when it accesses $para0$ via its original register index. To solve this problem, the value of register $v4$ is moved to register $v1$. The final stack frame is arranged as shown in Fig. 3c. The original registers are placed at the top of the frame, followed by an extra temporary register, and the taint tags of the original registers are stored in the shadow registers placed at the bottom of the stack frame.

4.1.2 Actual Parameters, Return Value and Exception

Actual parameters are also passed via the internal stack. Before invoking a method, the caller places the actual parameters on the top area of its stack frame, which is overlapped with the callee's stack frame, so that they will become the callee's formal parameters. A natural idea is to allocate taint tag storage in the overlapped area. However, this requires declaring additional parameters in the method prototype, which may result in extensive modifications of the original application instructions. In TaintMan, we take a different approach: place the taint tags of the caller's actual parameters in a global taint tag list. In this way, we keep the method prototype unchanged, which is very useful for not breaking features like reflection. The return value (if executed normally) and the thrown exception (if executed abnormally) of a method are stored in special internal variables maintained by the runtime. We also place their taint tags in the global taint tag list.

In order to support recursive method invocations, the taint tag list should be reusable at each call site and return site. To this end, at the entry of the callee, the taint tags of actual parameters are taken from the taint tag list and stored in the formal parameters' shadow registers. When the flow returns to the caller, the taint tag of the return value or the thrown exception is taken from the taint tag list and stored in the corresponding shadow register. As such, the taint tag list can be reused for the subsequent method invocations.

In a multi-threaded program, there may be more than one thread invoking the same method at the same time. As a result, the taint tag list may be simultaneously read or written by multiple threads. To ensure data consistency, the taint tag list should be thread-specific. In Java, each thread corresponds to a *Thread* instance and can be correlated with a *ThreadLocal* object to store thread-specific data. An intuitive way is to store the taint tag list for each thread in a *ThreadLocal* object. However, it is not very efficient, since accessing *ThreadLocal* storage involves method invocations and hash mappings. To this end, we store the taint tag list for each thread in an instance field additionally inserted into the *Thread* class, in which the access only requires a simple field operation.

4.1.3 Class Fields

Taint tag storage is allocated for each class field (static or instance) by inserting a shadow field into the class. There is a caveat that the Android runtime has some restrictions on the structure of certain system classes. Specifically, the runtime restricts that the *value* field of the *String* class should be placed at a fixed offset of the class structure and the wrapper classes (e.g., *Integer*) should have only one instance field. Inserting shadow fields into these classes may violate the restrictions, resulting in abnormal termination of the application. In order not to violate the field offset restrictions, shadow fields should be placed after all of the original fields. Since the class fields are arranged according to the alphabet order of field name, this can be achieved by prefixing the names of the shadow fields with the composition of the last element in the alphabet set (e.g., "zzz_"). To avoid violating the field number restrictions for wrapper classes, we create a new class named *TMObject* that contains the shadow field, and make it the super class of the wrapper classes. In this way, the wrapper classes can inherit the shadow field from *TMObject* without increasing the field number.

4.1.4 Arrays

We store only one taint tag per array to minimize storage overhead. In Dalvik, array is a built-in class. We cannot add an additional shadow filed to the array class, nor can we make the array class inherit from an extra super class containing shadow fields. To this end, we maintain a taint hash map between array objects and taint tags. A hash item is created only when an array is actually tainted. In Java, *String* is a frequently used class. The data of a *String* object is stored as a character array referenced by its *value* field. We define the taint of a *String* object as the taint of the character array referenced by its *value* field.

4.2 Tracking Data Dependence

TaintMan adopts the classic taint propagation logic for tracking data dependence: given an instruction, the taint value of its destination operand is set to the union of the taint values associated with its source operand(s). *TaintMan implements taint propagation logic by adding taint enforcement code into the target bytecode file itself, rather than into the runtime interpreter or emulator*. The instrumented taint enforcement code is written as Dalvik instructions. In general, the fewer taint enforcement instructions are used, the less overhead is incurred. Hence, it is expected to use as few instructions as possible to implement the taint propagation logic. The taint propagation logic for an instruction can be further refined to more specific ones with consideration of the relationship among operands. Generally, the refined taint propagation logic requires fewer taint enforcement instructions than the original one.

The taint propagation logic for tracking data dependence is shown in Table 1. In the table, $\tau(v)$ is the taint map returning the taint tag of variable v. For concise representation, we group the instructions with similar operational semantics into a single abstract instruction. Since the original implementation of TaintMan [21], we further refine the taint propagation logic for each kind of instruction and elaborately design appropriate taint enforcement code for the refined logic. In the rest of this section, we take the binary operation instruction and the field access instruction as examples to illustrate how to implement taint propagation logic for tracking data dependence.

4.2.1 Taint Propagation for Binary Operation

The operational semantic of the binary operation instruction *binary-op* v_A, v_B, v_C is $v_A \leftarrow v_B \otimes v_C$, meaning to perform the

TABLE 1
Taint Propagation Logic for Tracking Data Dependence

Rule	Instruction	Semantics	Coarse Taint Propagation Logic	Condition	Refined Taint Propagation Logic	Description
D1	*const-op* v_A, c	$v_A \leftarrow c$	$\tau(v_A) = 0$	N/A	$\tau(v_A) = 0$	Clear v_A taint
D2	*move-op* v_A, v_B	$v_A \leftarrow v_B$	$\tau(v_A) = \tau(v_B)$	A = B $A \neq B$	N/A $\tau(v_A) = \tau(v_B)$	Do nothing Set v_A taint to v_B taint
D3	*unary-op* v_A, v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) = \tau(v_B)$	A = B $A \neq B$	N/A $\tau(v_A) = \tau(v_B)$	Do nothing Set v_A taint to v_B taint
D4	*binary-op* v_A, v_B, v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) = \tau(v_B) \cup \tau(v_C)$	A = B = C $B = C \&\& A \neq B$ $A = B \&\& A \neq C$ $A = C \&\& A \neq B$ $A \neq B \neq C$	N/A $\tau(v_A) = \tau(v_B)$ $\tau(v_A)\| = \tau(v_C)$ $\tau(v_A)\| = \tau(v_B)$ $\tau(v_A) = \tau(v_B) \cup \tau(v_C)$	Do nothing Set v_A taint to v_B taint Combine V_C taint to v_A taint Combine v_B taint to v_A taint Set v_A taint to v_B taint $\cup$ v_C taint
D5	*sget-op* v_A, f	$v_A \leftarrow f$	$\tau(v_A) = \tau(f)$	N/A	$\tau(v_A) = \tau(f)$	Set v_A taint to field f taint
D6	*sput-op* v_A, f	$f \leftarrow v_A$	$\tau(f) = \tau(v_A)$	N/A	$\tau(f) = \tau(v_A)$	Set field f taint to v_A taint
D7	*iget-op* v_A, v_B, f	$v_A \leftarrow v_B.f$	$\tau(v_A) = \tau(v_B) \cup \tau(v_B.f)$	A = B $A \neq B$	$\tau(v_A)\| = \tau(v_B.f)$ $\tau(v_A) = \tau(v_B) \cup \tau(v_B.f)$	Combine $v_B.f$ taint to v_A taint Set v_A taint to object v_B taint $\cup$ field $v_B.f$ taint
D8	*iput-op* v_A, v_B, f	$v_B.f \leftarrow v_A$	$\tau(v_B.f) = \tau(v_A)$	A = B $A \neq B$	N/A $\tau(v_B.f) = \tau(v_A)$	Do nothing Set field $v_B.f$ taint to v_A taint
D9	*aget-op* v_A, v_B, v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) = \tau(v_B[\cdot]) \cup \tau(v_C)$	$A = B \&\& A \neq C$ $A = C \&\& A \neq B$ $A \neq B \neq C$	$\tau(v_A)\| = \tau(v_C)$ $\tau(v_A)\| = \tau(v_B[\cdot])$ $\tau(v_A) = \tau(v_B[\cdot]) \cup \tau(v_C)$	Combine index v_C taint to v_A taint Combine array $v_B[\cdot]$ taint to v_A taint Set v_A taint to array $v_B[\cdot]$ taint $\cup$ index v_C taint
D10	*aput-op* v_A, v_B, v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot])\| = \tau(v_A)$	N/A	$\tau(v_B[\cdot])\| = \tau(v_A)$	Combine v_A taint to array $v_B[\cdot]$ taint
D11	*return-op* v_A	$r \leftarrow v_A$	$\tau(r) = \tau(v_A)$	N/A	$\tau(r) = \tau(v_A)$	Set return value r taint to v_A taint
D12	*move-result-op* v_A	$v_A \leftarrow r$	$\tau(v_A) = \tau(r)$	N/A	$\tau(v_A) = \tau(r)$	Set v_A taint to return value r taint
D13	*invoke-op* $\bar{v}_{apar}, m$	$\bar{v}_{fpar} \leftarrow \bar{v}_{apar}$	$\tau(\bar{v}_{fpar}) = \tau(\bar{v}_{apar})$	N/A	$\tau(\bar{v}_{fpar}) = \tau(\bar{v}_{apar})$	Set formal param taints to actual param taints
D14	*throw-op* v_A	$e \leftarrow v_A$	$\tau(e) = \tau(v_A)$	N/A	$\tau(e) = \tau(v_A)$	Set exception e taint to v_A taint
D15	*move-exception-op* v_A	$v_A \leftarrow e$	$\tau(v_A) = \tau(e)$	N/A	$\tau(v_A) = \tau(e)$	Set v_A taint to exception e taint

Register variables are denoted as v_X, with X as the register index. Class static fields are denoted as f. Class instance fields are denoted as $v_Y.f$, where v_Y is an instance object reference. Array variables are denoted as $v_Z[\cdot]$, where v_Z is an array object reference. $\bar{v}_{fpar}$ and $\bar{v}_{apar}$ respectively stand for the formal parameter vector and the actual parameter vector. r is the return value and e is the thrown exception. c and m stand for a constant and a method respectively. $\tau(v)$ is the taint map returning the taint tag of variable v.

specified binary operation on the two source registers v_B and v_C, and store the result in the destination register v_A. The coarse taint propagation logic of the instruction is $\tau(v_A) = \tau(v_B) \cup \tau(v_C)$, which can be refined as shown in Rule D4 of Table 1. In particular, when the instruction takes the same register as its source and destination operands (i.e., $A = B = C$), no taint propagation is needed. By refining taint propagation logic in different conditions, we can simplify the taint propagation operation as much as possible.

In the case that the *binary-op* instruction takes different registers as its operands (i.e., $A \neq B \neq C$), the refined taint propagation logic is still $\tau(v_A) = \tau(v_B) \cup \tau(v_C)$. This propagation logic is primarily implemented by adding the bitwise-OR instruction *or-int* v_A', v_B', v_C' before the *binary-op* instruction. Here, v_A', v_B' and v_C' stand for the shadow registers of v_A, v_B and v_C respectively. Note that the *or-int* instruction requires all of its operand registers are indexed less than 256. If a certain shadow register is indexed higher than or equal to 256, a lower-indexed register needs to be selected as a substitute. The value of the shadow register should be moved forward to and back from the substitute register before and after the taint propagation.

According to different cases of the indexes of shadow registers, the taint propagation logic $\tau(v_A) = \tau(v_B) \cup \tau(v_C)$ can be implemented by different compositions of taint enforcement instructions, as shown in Table 2. Take Case 8 as an example. In this case, the shadow registers v_A', v_B' and v_C' are all indexed higher than or equal to 256, while the temporary register v_T and the destination operand v_A of the *binary-op* instruction are indexed less than 256. We select v_T and v_A as the substitute registers for v_B' and v_C' respectively, and reuse v_A as the substitute register for v_A'. Since the current values of v_T and v_A have no effect on the subsequent instructions after the instrumentation point, there is no need to save and restore their values. The taint enforcement instructions for Case 8 are composed of four taint enforcement instructions, which occupy 17 bytes.

4.2.2 Taint Propagation for Field Operation

The operational semantic of the field access instruction *iput-op* v_A, v_B, f is $v_B.f \leftarrow v_A$, meaning to store the value of the source register v_A to the instance field f of the specified object in the destination register v_B. The coarse taint propagation logic of the instruction is $\tau(v_B.f) = \tau(v_A)$, which can be refined as shown in Rule D8 of Table 1. In particular,

TABLE 2
Enforcement of $\tau(v_A) = \tau(v_B) \cup \tau(v_C)$ in Different Cases

Case	Condition	Taint Enforcement Instructions (TEIs)	TEIs Count	TEIs Bytes
1	$A' < 256$ && $B' < 256$ && $C' < 256$	*or-int* v_A', v_B', v_C'	1	4
2	$A' < 256$ && $B' \geq 256$ && $C' < 256$	*move/from16* v_A', v_B' *or-int* v_A', v_A', v_C'	2	8
3	$A' < 256$ && $B' < 256$ && $C' \geq 256$	*move/from16* v_A', v_C' *or-int* v_A', v_A', v_B'	2	8
4	$A' \geq 256$ && $B' < 256$ && $C' < 256$	*or-int* v_A, v_B', v_C' *move/16* v_A', v_A	2	9
5	$A' < 256$ && $B' \geq 256$ && $C' \geq 256$	*move/from16* v_A', v_B' *move/from16* v_A, v_C' *or-int* v_A', v_A', v_A	3	12
6	$A' \geq 256$ && $B' \geq 256$ && $C' < 256$	*move/from16* v_A, v_B' *or-int* v_A, v_A, v_C' *move/16* v_A', v_A	3	13
7	$A' \geq 256$ && $B' < 256$ && $C' \geq 256$	*move/from16* v_A, v_C' *or-int* v_A, v_A, v_B' *move/16* v_A', v_A	3	13
8	$A' \geq 256$ && $B' \geq 256$ && $C' \geq 256$ && $T < 256$	*move/from16* v_T, v_B' *move/from16* v_A, v_C' *or-int* v_A, v_T, v_A *move/16* v_A', v_A	4	17
9	$A' \geq 256$ && $B' \geq 256$ && $C' \geq 256$ && $T \geq 256$	*move/from16* v_A, v_B' *move/16* v_T, v_C *move/from16* v_C, v_C' *or-int* v_A, v_A, v_C *move/16* v_A', v_A *move/from16* v_C, v_T	6	22

when the instruction takes the same register as its source and destination operands (i.e., $A = B$), no taint propagation is needed, since the taint to be attached to the instance field is implicated in the taint of the identified object.

In the case that the *iput-op* instruction takes different registers as its operands (i.e., $A \neq B$), the refined taint propagation logic is still $\tau(v_B.f) = \tau(v_A)$. This propagation logic is primarily implemented by adding the field setting instruction *iput* v_A', v_B, f' before the *iput-op* instruction. Here, v_A' and f' respectively stand for the shadow register of v_A and the shadow field of f. Note that the *iput* instruction requires all of its operand registers are indexed less than 16. Table 3 shows the implementation of the taint propagation logic $\tau(v_B.f) = \tau(v_A)$ with consideration of the indexes of shadow registers. Take Case 3 as an example, where the shadow register v_A' and the temporary register v_T are indexed higher than or equal to 16. We select v_A as the substitute register for v_A'. Since v_A may be used after the instrumentation point, we need to save and resume its value before and after the taint propagation. The taint enforcement instructions for Case 3 are composed of four taint enforcement instructions, which occupy 12 bytes.

4.3 Tracking Strict Control Dependence

Compared with the explicit information flow, the implicit information flow is more difficult to track. In general, it is intractable to perform sound and complete tracking of IIF [5]. In TaintMan, we mitigate the threat of IIF by developing a tracking algorithm based on our prior work [11] (a short paper). The algorithm tracks a special kind of control dependence called *strict control dependence*, whose nature highly resembles that of data dependence and hence is most likely to be leveraged for attacks. The basic idea of the SCD tracking algorithm is to selectively taint a predicate if it has strong correlation with sensitive information. All assignments guarded by such a tainted predicate are tainted. For better efficiency, we adopt a lazy tainting policy, which postpones the tainting of control dependence to the post-dominator of a control structure.

TABLE 3
Enforcement of $\tau(v_B.f) = \tau(v_A)$ in Different Cases

Case	Condition	Taint Enforcement Instructions (TEIs)	TEIs Count	TEIs Bytes
1	$A' < 16$	iput v_A', v_B, f'	1	3
2	$A' \geq 16$ && $T < 16$	*move/from16* v_T, v_A' *iput* v_T, v_B, f'	2	6
3	$A' \geq 16$ && $T \geq 16$	*move/from16* v_T, v_A *move/from16* v_A, v_A' *iput* v_A, v_B, f' *move/from16* v_A, v_T	4	12

4.3.1 Identifying SCD Branches

A statement s is strictly control dependent on a predicate p with v_p the predicate variable, if the execution of s can precisely infer the value of v_p. The branch leading to the execution of s is called the SCD branch. Consider the example in Fig. 4a. If the assignment statement at line 03 is executed, the attacker can precisely infer the value of the predicate variable *secret* in predicate at line 02 is 1. Thus, there is an SCD between the assignment statement and the *if* statement. The true branch of the *if* statement is the SCD branch. As a counter example, consider the code snippet in Fig. 4b. There is control dependence between the assignment statement at line 03 and the *if* statement at line 02. However, from the execution of the assignment statement, we can only infer that the predicate variable *secret* is larger than 1. Little information is revealed. Thus, this control dependence is not SCD.

The first step of SCD tracking is to identify SCD branches through static analysis. Currently, our analysis only concentrates on SCDs caused by equivalence testing. It first considers the common conditional structures. For an *if* structure, if it is an equivalence test (i.e., ==), the true branch is an SCD branch; or if it is a non-equivalence predicate (i.e., !=), the false branch is an SCD branch. For a *switch* structure, if a branch can be reached from only one *case* value, it is an SCD branch. We also handle other IIF-inducing control structures (e.g., exception-prone instructions and polymorphic method invocations) by explicitly converting them into either the *if* structure or the *switch* structure, depending on the number of their branches. The explicit *if* or *switch* statements are further instrumented to track SCD.

```
01 public = 0;
02 if (secret == 1) {
03     public = 1;
04 }
05 output(public);
```

(a). SCD

```
01 public = 0;
02 if (secret > 1) {
03     public = 1;
04 }
05 output(public);
```

(b). non-SCD

Fig. 4. Example of SCD and non-SCD.

Rule	Event	Instrumentation
C1	Encountering a predicate statement: x is the predicate variable.	$t = \tau(x)$;
C2	Encountering a post-dominator: variable x is attached with $\Phi(x_1, \ldots, x_n)$; $\neg\exists x_i, x_j$: x_i corresponds to an SCD branch and $i \neq j$ and $\lvert x_i\rvert = \lvert x_j\rvert$.	$\tau(x) = \tau(x) \mid t$;
C3	Encounter a statement s: s is contained in an SCD branch; s may immediately propagate the value outward to a global variable x.	$\tau(x) = \tau(x) \mid t$;

Fig. 5. Rules for lazy strict control dependencee tainting. $|x_i|$ denotes the value of variable x of version i in SSA representation.

4.3.2 Lazy Tainting Policy

For better performance, we propose a lazy tainting policy, instead of on-the-fly tainting. The lazy tainting policy is implemented with the information of CFG, PDT and SSA computed during the static analysis phase. More specifically, we leverage CFG and PDT to determine the effective scope of a control structure, and leverage SSA to understand the assignments of variables in different branches. In SSA representation, each assignment of a variable generates a new version for the variable. If a variable has different assignments in different branches, a Φ function is added at the merge node (i.e., the immediate post-dominator), which lists the versions of the variable along each branch.

The rules for lazy strict control dependence tainting are shown in Fig. 5. When encountering a predicate statement, the taint of the predicate variable is stored in a temporary variable (Rule C1). At each immediate post-dominator, the algorithm examines each variable that is assigned in an SCD branch to check whether its value is distinctive from other branches. If so, it will propagate the taint of the current control structure to the identified variable (Rule C2).There is a special consideration for the lazy tainting policy. An SCD branch may contain some assignment statements whose values may immediately escape the branch (e.g., assignments to global variables). When encountering these statements in an SCD branch, we propagate taints immediately (Rule C3).

4.4 Optimization

To reduce performance overhead, we enforce on-demand instrumentation (before execution) and on-demand tracking (at runtime) to avoid unnecessary taint analysis whenever possible. In such a two-staged optimization solution, we first preliminarily identify the potential taint-related methods in a light-weight fashion, then make a precise tracking decision depending on the taint situation at runtime. It enables us to achieve the same optimization goal as [22] without heavy and complex global static data flow analysis.

4.4.1 On-Demand Instrumentation

With regard to taint analysis, methods can be divided into three categories: source-related APIs that can introduce new taints into the application, sink-related APIs that can transmit taints out of the application, and other methods that can only propagate taints within the application. A method needs to be instrumented only if it may exist in an execution path from a source-related API to a sink-related API. For programs written in a Java-like feature-rich language, it is very difficult, if not impossible, to identify all possible source-to-sink paths. To this end, we propose a conservative approach based on the observation: a method may exist in a source-to-sink path, only if it could import tainted data from and export tainted data to the outside of the method.

TABLE 4
Effect of Dalvik Instructions on Data

Instruction	Semantics	Effects on Data	Type
const-op v_A, c	$v_A \leftarrow c$	Reset data	DPI
move-op v_A, v_B	$v_A \leftarrow v_B$	Propagate within method	DPI
unary-op v_A, v_B	$v_A \leftarrow \otimes v_B$	Propagate within method	DPI
binary-up v_A, v_B, v_C	$v_A \leftarrow v_B \otimes v_C$	Propagate within method	DPI
sget-op v_A, f	$v_A \leftarrow f$	Import from static field	DII
iget-op v_A, v_B, f	$v_A \leftarrow v_B.f$	Import from instance field	DII
aget-opv_A, v_B, v_C	$v_A \leftarrow v_B[v_C]$	Import from array	DII
move-result-op v_A	$v_A \leftarrow r$	Import from return value	DII
move-exception-op v_A	$v_A \leftarrow e$	Import from exception site	DII
sput-op v_A, f	$f \leftarrow v_A$	Export to static field	DEI
iput-op v_A, v_B, f	$v_B.f \leftarrow v_A$	Export to instance field	DEI
aput-op v_A, v_B, v_C	$v_B[v_C] \leftarrow v_A$	Export to array	DEI
return-op v_A	$r \leftarrow v_A$	Export to return value	DEI
invoke-op $\bar{v}_{apar}, m$	$\bar{v}_{fpar} \leftarrow \bar{v}_{apar}$	Export to callee	DEI
throw-op v_A	$e \leftarrow v_A$	Export to exception handler	DEI

In Android, data importation and exportation of a method have to be conducted by passing parameters or executing special Dalvik instructions. We analyze the Dalvik bytecode language to learn the effect of instructions on data. The analysis result is shown in Table 4. According to their effects on data, the Dalvik instructions can be categorized into three types: (1) data-propagation instructions (DPIs) that propagate data within the method; (2) data-importation instructions (DIIs) that import data from the outside of the method; and (3) data-exportation instructions (DEIs) that export data to the outside of the method.[4]

Note that in addition to normal data flow, taints can also be propagated through exceptional data flow. The *throw-op* instruction exports data as an exception object to the exception handler, and the *move-exception-op* instruction imports data as an exception object from the exception site. The exception site and the exception handler may locate across the method boundary, and hence we treat *throw-op* instruction as an DEI and *move-exception-op* as an DII.

Based on the above analysis, we believe that a method needs to be instrumented only if it satisfies the following two conditions: (1) It has parameter(s) or contains at least one DII; and (2) It contains at least one DEI. A light-weight static analysis is performed before the instrumentation to identify the methods satisfying both of the above conditions. Only the identified methods are instrumented.

4.4.2 On-Demand Tracking

Even if a method could propagate taints across the method scope (thus needs to be instrumented), it will not always propagate taints on each execution instance. Indeed, only when a method actually imports taints at runtime, can it actually propagate taints. It means that an instrumented method does not need to be tracked until the first time it imports taints from the outside. Based on this idea, we propose the on-demand tracking technique to reduce the

4. DIIs and DEIs can be treated as method-local sources and sinks proposed in [23]. In TaintMain, we focus on their effect on data propagation for optimization purpose.

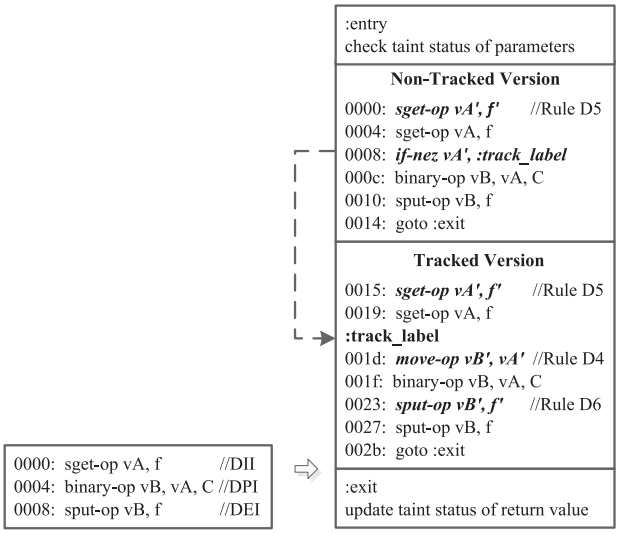


Fig. 6. Instrumentation of method *Logger$Stream.endIndent()* for enforcing on-demand tracking. Taint tags of register v_X and field f are denoted as v_X' and f' respectively.

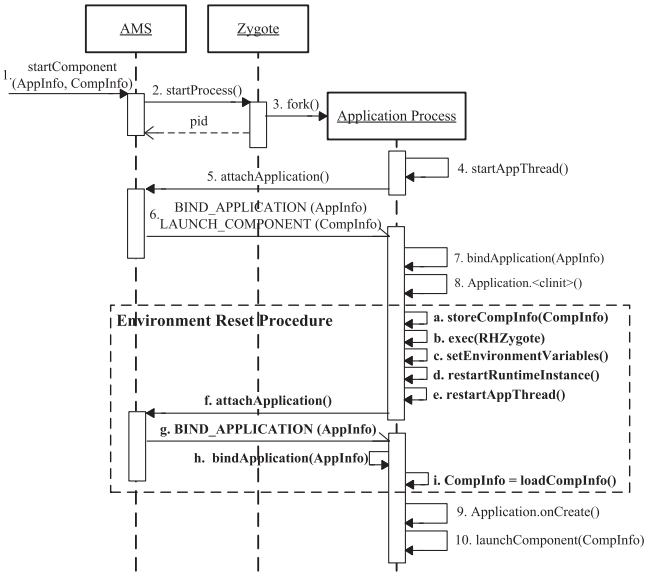


Fig. 7. Sequence diagram of the application startup process. Additional steps introduced by the environment reset procedure are outlined with a dashed rectangle.

runtime overhead. For each instrumented method, there are two versions of its 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.

The non-tracked version and the tracked version coexist in the instrumented method.[5] In the non-tracked version, only DIIs are monitored. Specifically, a conditional transfer instruction is added after each DII to transfer the control to the tracked version in the case that the imported data is tainted. The conditional transfer instruction uses a symbolic address (i.e., label) as its target address to avoid the complex offset computation. In the tracked version, all DPIs, DEIs and DIIs are monitored with taint enforcement code. For ease of transfer from the non-tracked version, we introduce a label after each DII. 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 system class method *Logger$Stream.endIndent*() 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 the static field f. As shown in Fig. 6, the original method bytecode contains *sget-op*, *binary-op*, *sput-op* and *return-void* instructions. After instrumentation, there are two versions of the 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 the *sget-op*, *binary-op*, and *sput-op* instructions are monitored with taint enforcement 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 transfer from Offset 0x0008 in the non-tracked version to Offset 0x001d in the tracked version.

5. The tracked version is placed next to the non-tracked version. To ensure the instrumented method has a single entry and a single exit, we wrap it with an operation at the start to check the taint of parameters, and an operation at the end to update the taint of the return value.

4.5 Reference Hijacking

The goal of reference hijacking is to take control over the reference of the target application to the underlying system class libraries, so that it can be redirected to the instrumented alternatives.[6] It is mainly achieved via a special environment reconstruction procedure. By modifying the application startup process, additional operations are introduced to drive the application to restart with new environment variables (e.g., the library path). After restarting, the target application will be executed in a new execution environment, in which the instrumented system class libraries are referenced instead of the original ones.

Fig. 7 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). As 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 in sequence. Next, the target application is instantiated (Step 7) to execute the class initialization method (Step 8) and the *onCreate*() method (Step 9) of the Application class. Finally, the target component is launched (Step 10).[7]

6. System libraries are the core of the runtime environment, which can be fully taken control of by reference hijacking. In addition to system libraries, the runtime environment includes the OS kernel and system services, which are beyond the control of reference hijacking.

7. The *onCreate*() method of an Activity, a Service or a Broadcast Receiver is called after the *onCreate*() method of the *Application* class. For a Content Provider, its *onCreate*() method is called before the *onCreate*() method of the *Application* class, but after the class initialization method of the *Application* class.

In practice, manufacturers often modify some aspects of Android. However, as a fundamental feature of Android, the application startup process is most likely to remain unchanged. According to our empirical evaluation on a variety of devices from prevalent manufacturers (e.g., Samsung, LG, Motorola, HUAWEI, etc.) and most popular custom ROMs (e.g., CyanogenMod, MIUI, etc.), we found that none of their customizations modify the application startup process.

The environment reconstruction procedure is invoked between Step 8 and Step 9. After Step 8, the customized Application class RHApplication 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. Thus, we store it in a temporary file. Step b: execute the RHZygote program file. 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 class libraries is cut off. The RHZygote program performs Step c to Step e to prepare a new execution environment. Step c: set environment variables. Specifically, the `BOOTCLASSPATH` environment variable is set to specify the instrumented system class libraries as the default class paths. Step d: start a new runtime instance. The new runtime instance will load system class libraries from the paths specified by the aforementioned environment variable. Step e: restart an application thread. The restarted application thread will interact with AMS to reload the target application and instantiate its RHApplication class for a second time (Step f to Step h). At this time, RHApplication performs Step i to obtain the information of the target component from the temporary file. Finally, the target application will finish the initialization at Step 9 and launch the target component for execution at Step 10.

5 Evaluation

We have developed TaintMan and successfully deployed it on two prevalent devices. One is HTC One S (Android 4.0.4, Dalvik). The other is Motorola Moto G (Android 5.0.2, ART).

5.1 Effectiveness Evaluation

The effectiveness evaluation is performed with three sets of applications. The first set is malware samples selected from the Android Malware Genome Project dataset [14]. This malware dataset contains 49 malware families, of which 26 are known to steal user's private information [4]. To seek a trade-off between the test coverage and the test effort, we randomly select 20 samples from each malware family (if it has), generating 150 malware samples. The second set is real-world applications collected from the official Android market (i.e., Google Play [15]), the HTC vendor's market (HTC Store [16]), and two third-party markets (App China [17] and Slide Me [18]). From each market, we select 25 recently released free applications from the global popularity list, generating 100 application samples. The third set contains nine proof-of-concepts (PoCs) from our prior work [11] and two real-world malware samples that leverage IIF. As a comparison, we also deploy TaintDroid on the emulator. For each tested target, we execute it in both TaintMan and TaintDroid. We should note that both TaintMan and TaintDroid do not distinguish benign privacy leakage (i.e., necessary for normal functionalities) from malicious privacy leakage, and treat any privacy leakage as suspicious.

5.1.1 Privacy Leakage Detection in Malware Samples

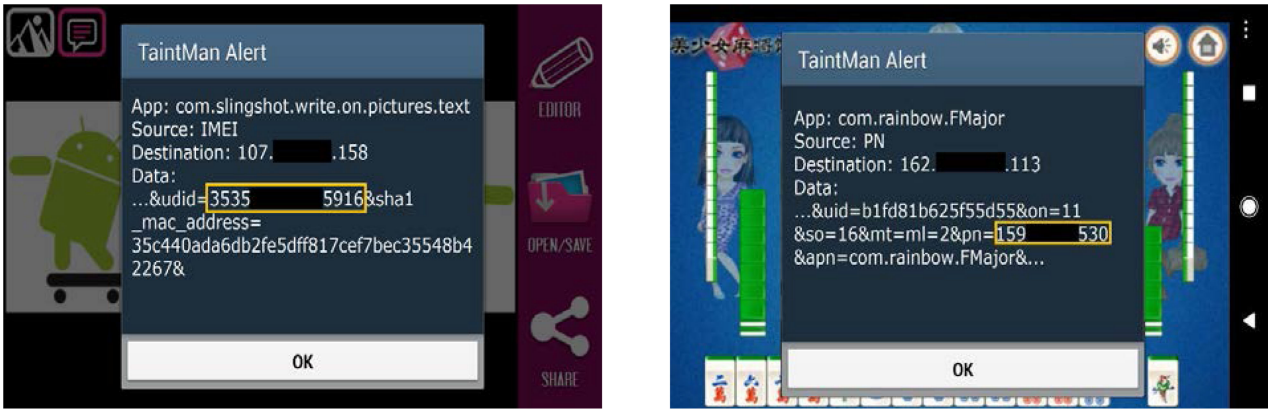
TaintMan report that all of the 150 malware samples leak user's private information. Among them, 84 samples are found to leak more than one kind of private infromation. IMEI+PN is the most common combination of private information leaked by the malware samples. We use TaintDroid to confirm the detection results of TaintMan, and found that none of them is false alarm. We should highlight the detection of three malware samples from the *DroidKungFu3* family. The privacy-breaching behavior of these samples will not be triggered in the emulator. Specifically, they obtain the IMEI string of the infected device, and decide whether to send it to a remote server by judging whether the IMEI string is all 0's (which means it is running in an emulator). Fortunately, when the three samples are analyzed by TaintMan deployed in the real device, their malicious behaviors are triggered and can be detected. We list the detailed detection result in the supplemental material, MalwareDetection.pdf.

5.1.2 Privacy Leakage Detection in Real-World Apps

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 the four applications and found they indeed access private information (i.e., invoking the corresponding APIs) and send them to a remote server (which are captured by Tcpdump [24]). Take one of them, Write on Pictures, as an example. As shown in Fig. 8a, 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. We should highlight that besides the four applications that cannot run on TaintDroid, there are two applications that can run on TaintDroid only when the JIT compilation mode is disabled. It is probably caused by the incompatibility due to modification of the underlying system. The issue is shared by the existing dynamic instrumentation approaches [7], [8]. We list the detailed detection result in the supplemental material, ApplicationDetection.pdf.

5.1.3 Implicit Information Flow Detection

The nine PoCs of our prior work [11] have been developed based on a (potentially malicious) game application FMajor [25]. During execution, FMajor obtains the host device's phone number, propagates it via explicit information flow (e.g., assignments), and finally sends it out via Internet. We modify the procedure of phone number transmitting, inserting an additional transformation step. This step employs a certain IIF form to transform each character of the phone



(a) Privacy Leakage via Excplicit Information Flow (b) Privacy Leakage via Implicit Information Flow

Fig. 8. TaintMan can successfully detect privacy leakage via explicit information flow and implicit information flow.

number string. After the transformation step, the transformed string will hold the same value as the original phone number string, although it is not achieved by data dependence but rather IIF. TaintMan can detect the privacy leakage behaviors of all these PoCs. Fig. 8b shows a screenshot of the detection. As a comparison, TaintDroid cannot detect these PoCs, since it ignores the implicit information flow. We also evaluate TaintMan with two real-world malware samples from the *DroidKungFu3* and *AnserverBot* family that leverage IIF.[8] TaintMan can successfully detect the privacy leakage behaviors.

5.2 Performance Evaluation

5.2.1 Evaluation with Standard Benchmark

We use CaffeineMark [26] to evaluate the performance overhead of TaintMan. CaffeineMark is a famous benchmark, which 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. For concise illustration, here we only present the performance evaluation on the Motorola Moto G.

First, we evaluate the performance improvement by the refined taint logic. In this evaluation, we disable on-demand instrumentation and on-demand tracking optimizations, and measure the instrumentation impact of the *binary-op* instructions and the *iput-op* instructions individually. We use the *Loop* test of CaffeineMark as a metric, which contains *binary-op* and *iput-op* instructions in around 2,048 iterations. Before refining, all *binary-op* and *iput-op* instructions are instrumented as in the worst case (i.e., Case 9 of Table 2 and Case 3 of Table 3). The performance improvements are 11 percent for *binary-op* instructions (11,704 before refining versus 12,992 after refining) and 6 percent for *iput-op* instructions (22,141 before refining versus 23,470 after refining).

Then, we evaluate the performance improvement by optimizations. The evaluation result is shown in Fig. 9. We can see that the overall performance overhead incurred by data dependence and strict control dependence tracking is 42.3 percent (13,689 before instrumentation versus 7,898 after instrumentation without optimization). The on-demand instrumentation alone improves the performance by 9 percent (7,898 before enforcing on-demand instrumentation versus 8,641 after enforcing). The combination of on-demand instrumentation and on-demand tracking improves the performance by 23 percent (7,898 without optimization versus 9,737 with optimizations). After optimization, the performance overhead is acceptable for analysis purpose.

5.2.2 Evaluation with Real-World Applications

We choose 10 real-world applications of different categories to evaluate the impact of TaintMan on the interactive applications. We use application load time delay, Activity launch time delay and user input response delay as metrics. The result shows TaintMan incurs 980 ms application load time delay, 320 ms Activity launch time and 170 ms user input response delay. That means TaintMan has no noticeable interference on the interactive behaviors of applications.

We also evaluate the space overhead of TaintMan. We measure the sizes of the 10 real-world applications and the sizes of system class libraries of HTC One S and Motorola MOTO G before and after instrumentation respectively. The result shows that the instrumentation process only increases the application size by 23 percent (4.0 MB avg. versus 4.9 MB avg.). In addition, the instrumentation process

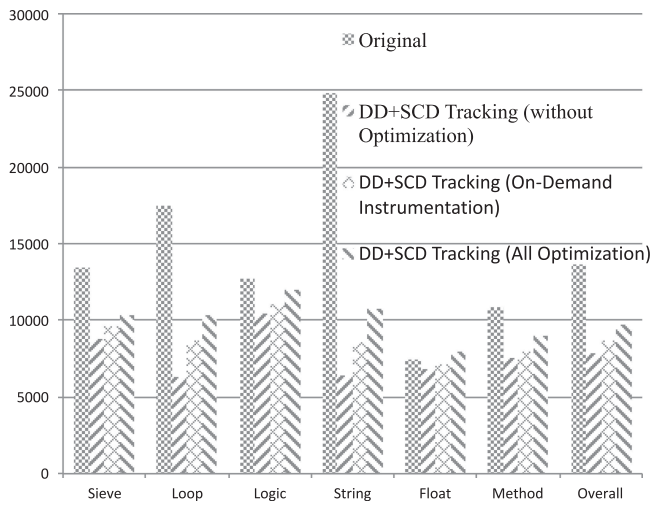


Fig. 9. Performance evaluation with CaffeineMark.

8. The *DroidKungFu3* malware sample (MD5: 1d908963aa08e265190817f88bb3ae3c) leverages IIF to encode the ICCID string. The *AnserverBot* malware sample (MD5: c7d856feaab717913889e175105873cd) leverages IIF to encode the IMEI string.

approximately triples the total size of system class libraries (15.8 MB versus 42.7 MB for HTC One S, 50.7 MB versus 127.3 MB for Motorola MOTO G). Since system class libraries are shared by all instrumented applications and the size increment is relatively small compared with the large pool of applications, we argue the size overhead is acceptable.

6 Discussion

Compatibility with ART. The instrumentation of TaintMan is conducted at the bytecode level. The way we allocate taint tag storage and enforce taint propagation logic ensures the instrumented code to be valid as if it were generated normally from Java source code. For example, the allocation of tag taint for field f is equivalent to defining a new field f' in the class. The tracking code for field access instruction on field f is equivalent to executing another field access instruction on field f'. For on-demand tracking, the two versions of bytecode act as two branches of the operation checking whether any parameter is tainted. They are wrapped to have a single entry and a single exit, just like a normal method (see Fig. 6). Such instrumentation is transparent to the underlying mechanism (e.g., optimizations) of the ART compiler and will not be affected even if the ART compiler is updated in the future.

Instrumentation at Bytecode Level. Dynamic taint analysis can be implemented at different levels, including source-level, runtime-level, library-level and bytecode-level [27]. In TaintMan, we choose to implement the instrumentation at the bytecode level. The reasons are as follows. First, for a majority of real-world applications, we can only get their bytecode, hence cannot implement the instrumentation at the source level that requires source code. Second, we expect TaintMan to be easily deployable on real devices, hence cannot implement the instrumentation at the runtime level that requires modifying or rooting devices. Third, sensitive data can be propagated via system libraries as well as the target application code, hence it would be insufficient if we only implement the instrumentation at the library level.

Handling Dynamically Loaded Code. Android provides the dynamic loading mechanism to allow applications to load the Dalvik bytecode at runtime. TaintMan-like static-instrumentation-based DTA approaches fall short in handling dynamically loaded code. To mitigate, we wrap the dynamic loading APIs, such as *DexFile.openDexFile*(), to allow the analysts to obtain the original bytecode file to be loaded and specify an instrumented counterpart (generated off-line) as the substitute of the original bytecode file. In the future, we plan to port the instrumentation process into smartphone device, so that the dynamically loaded code can be instrumented on-the-fly.

Handling Integrity Checking. When an application is instrumented, its integrity is inevitably destroyed. Some applications may validate the integrity of their packages at runtime. For such applications, extra efforts need to be taken to bypass the integrity checking. In theory, complex anti-reverse-engineering would make the integrity checking robust against bypassing. However, our preliminary study on the integrity checking mechanism of real-world applications shows that the most common ways to validating the integrity of an application are examining its signature by querying from the Package Manager Service (PMS) via inter-process communication (IPC) calls, and examining the checksums of some critical files (e.g., the digest file MANIFEST.MF) by invoking the corresponding framework APIs. Since reference hijacking allows us to take full control over the reference of the target application to the underlying system libraries, we can redirect these IPC calls and APIs, so that a forged value will be returned as if the operations were carried on the original application package. As such, we can bypass the integrity checking.

Combining Reference Hijacking with Boxify. Reference hijacking requires to repackage the target application, hence suffers from issues with re-signing applications. An ideal way to avoid the problem is to combine reference hijacking with a very recent work, Boxify [28]. 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 top of security-enhanced underlying system libraries. We believe that the combination of reference hijacking and Boxify can extend the capability of both of them, and will create a wonderful solution for securing Android applications. We have begun to research how to leverage Boxify to further improve the practicability of the reference hijacking technique.

Extensibility to End Users. TaintMan is originally designed for analysts to track the sensitive information flow in Android applications. It overcomes a lot of problems that have affected wide-spread deployment. With some additional efforts, it could be extended to be more friendly to end users. For example, we can port the instrumentation component of TaintMan to the smartphone device to get rid of the off-line instrumentation step, as done in [29]. We can further extend TaintMan to declassify sensitive data automatically, rather than simply raising alarms, as done in [30].

Soundness of Implicit Information Flow Tracking. The lazy tainting policy for IIF tracking improves not only performance, but also precision. Consider the code snippet "public = 0; if (secret != 0) public = 1;". Variable *public* is initialized to 0. At runtime, if the false branch is taken (i.e., *secret* == 0), *public* will not be assigned. As a result, the taint of predicate variable *secret* will not be directly propagated to *public*. If we adopt the on-the-fly tainting policy, the dependence of *public* on *secret* could not be captured due to the execution omission. This issue can be addressed if we adopt the lazy tainting policy. By postponing tainting to the post-dominator of a control structure, we can have clear information about the assignments of variables in different branches via observing their Φ function. In the aforementioned example, variable *public* is attached with a Φ function listing two versions: one indicates that *public* takes a new value along the true branch, and the other indicates that *public* remains the same with its initialized value along the false branch. Given the assignment information, we can precisely propagate the taint of *secret* to *public* even if no assignment is executed at the false branch.

Currently, our IIF tracking algorithm only focuses on strict control dependence caused by equivalence testing (i.e., '==' or '!='), which is the common case we found in

in-the-wild malware. We do not take into account the domain of a variable, which could also induce strict control dependence. Consider the code snippet "if (secret >1) public = 1;". If variable *secret* can only ever have the values −2, 0 and 3 (for whatever reasons), then the value of variable *public* (i.e., 1) precisely determine the value of *secret* (i.e., 3). Identifying such strict control dependence requires the help of constraint solvers at the cost of performance overhead.

So far, our IIF tracking algorithm only tracks the innermost strict control dependence. To support tracking nested strict control dependence, a stack of implicit flow labels should be maintained: the taint label is pushed into the stack when encountering a predicate statement and popped out when encountering the post-dominator. This would have a high performance impact. Since we have not yet found Android malware leveraging nested strict control dependence, we only track the innermost strict control dependence for performance consideration.

7 Related Work

On the Android platform, many approaches have been proposed based on DTA. TaintDroid [6] and DroidScope [7] are the two representative works. TaintDroid provides a real-time system-wide information-flow tracking by instrumenting the Dalvik virtual machine interpreter, making it generate taint enforcement code for the executed instructions at runtime. Many malware detection systems (e.g., AppFence [8] and DroidBox [9]) are derived from TaintDroid. Since the latest Android system introduces ART as a substitute of the DVM runtime, the interpreter-instrumentation-based solutions are not applicable anymore. DroidScope instruments the Android emulator, rather than modifying the Dalvik interpreter. Although compatible with ART, it is limited in analyzing applications in the emulator, not in real devices.

Some state-of-the-art DTA approaches have paid attention to the ART runtime. DroidForce [31] uses instrumentation to enforce policies including data flow on unmodified phones. It only tracks inter-component flows dynamically and relies on statically pre-computed data flow tables for intra-component tracking. ARTist [23] and TaintART [32] add the taint enforcement code by modifying the optimizing backend used by the ART compiler for code generation. Their instrumentations are performed on the device, which requires rooting to deploy the instrumented ART compiler.

Recent years, attentions are paid to the deployability of security analysis approaches. Some attempts are made to enforce application-wide security features without flashing or rooting devices. For example, Aurasium [33] 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 [28] proposes a novel technique to enforce 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.

Styp-Rekowsky et al. [34] proposed a technique similar to reference hijacking. Their approach diverts the control flow towards the security monitor by modifying references to security-relevant methods in the Dalvik virtual machine, which does not require restarting the complete execution environment. This approach would incur minimal runtime overhead if the amount of security-relevant methods is small. For dynamic taint analysis, there are a considerable number of system methods needed to be tracked. Diverting all these method calls would take longer time than restarting the complete execution environment.

8 Conclusion

In this paper, we present TaintMan, an ART-compatible dynamic taint analysis framework that can be conveniently deployed on unmodified and non-rooted Android devices. With TaintMan, taint enforcement code is statically instrumented into both the target application and the system class libraries to track data flow and common control flow. A specially designed execution environment reconstruction procedure is introduced to force the target application to reference the instrumented system libraries. To improve the performance, we perform several optimizations. Specifically, we enforce on-demand instrumentation and on-demand tracking to avoid unnecessary taint analysis whenever possible. In addition, we design efficient taint tag storage and refine taint propagation logic to implement taint tracking with as little code as possible. We evaluate TaintMan with malware samples and real-world applications. The evaluation result shows that TaintMan can effectively detect privacy leakage behaviors. In addition, the performance overhead of TaintMan is acceptable for analysis purposes. We believe that TaintMan is a practical DTA framework applicable for the latest Android system on real devices.

Acknowledgments

The authors would like to thank the reviewers and editors for their constructive comments and suggestions. This research work was supported, in part, by National Natural Science Foundation of China (NSFC) under grants 91418206, 61170240 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. Bin Liang is the corresponding author.

References

[1] IDC, "Smartphone OS market share." 2016. [Online]. Available http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] F-Secure, "Threat report H2." 2013. [Online]. Available: http://www.f-secure.com/documents/996508/1030743/Threat_Report_H2_2013.pdf
[3] K. Chen et al., "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *Proc 24th USENIX Security Symp. USENIX Security* 2015, pp. 659–674.
[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. 33th IEEE Symp. Security Privacy* 2012, pp. 95–109.
[5] 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 *Proc. 31st IEEE Symp. Security Privacy*. 2010, pp. 317–331.

[6] W. Enck, et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation* 2010, pp. 393–407.
[7] L. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proc. 21th USENIX Security Symp.* 2012, Art. no. 29.
[8] 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 *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 639–652.
[9] DroidBox. [Online]. Available: https://github.com/pjlantz/droidbox
[10] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proc. 9th ACM Symp. Inf. Comput. Commun. Security*, 2014, pp. 447–458.
[11] W. You, B. Liang, J. Li, W. Shi, and X. Zhang, "Android implicit information flow demystified (short paper)," in *Proc. 10th ACM Symp. Inf. Comput. Commun. Security*, 2015, pp. 585–590.
[12] J. Schütte, D. Titze, and J. M. D. Fuentes, "AppCaulk: Data leak prevention by injecting targeted taint tracking into Android apps," in *Proc. 13th IEEE Int. Conf. Trust Security Privacy Comput. Commun.*, 2014, pp. 370–379.
[13] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for Android applications without firmware modding," in *Proc. 9th ACM Symp. Inf. Comput. Commun. Security*, 2014, pp. 259–270.
[14] Y. Zhou and X. Jiang, "Android malware genome project." (2015). [Online]. Available: http://www.malgenomeproject.org/
[15] Google Play (2016). [Online]. Available: https://play.google.com
[16] HTC Store. (2016). [Online]. Available: http://www.htc.com
[17] App China. (2016). [Online]. Available: http://www.appchina.com
[18] Slide Me. (2016). [Online]. Available: http://slideme.org
[19] Smali and BakSmali. (2016). [Online]. Available: https://github.com/JesusFreke/smali
[20] Apktool. (2016). [Online]. Available: http://ibotpeaches.github.io/apktool
[21] W. You, et al., "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 959–970.
[22] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp.*, 2014, pp. 45–61.
[23] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. 2nd IEEE Eur. Symp. Security Privacy* 2017, pp. 481–495.
[24] Android Tcpdump. (2016). [Online]. Available: http://www.androidtcpdump.com/
[25] FMajor. (2016). [Online]. Available: http://apps.lidroid.com/apks/65327/
[26] CaffeineMark. (2016). [Online]. Available: http://www.benchmarkhq.ru/cm30
[27] B. Livshits, "Dynamic taint tracking in managed runtimes," Microsoft Research, Tech. Rep. MSR-TR-2012-114, 2012.
[28] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock Android," in *Proc. 24th USENIX Security Symp.*, 2015, pp. 691–706.
[29] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - enforcing user requirements on android apps," in *Proc. 19th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 543–548.
[30] B. Livshits and S. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," in *Proc. 40th Annu. ACM SIGPLAN-SIGACT Symp. Principles Programming Languages*, 2013, pp. 385–398.
[31] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "DroidForce: Enforcing complex, data-centric, system-wide policies in Android," in *Proc. 9th Int. Conf. Availability Reliability Security*, 2014, pp. 40–49.
[32] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android runTime," in *Proc. 23rd ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 331–342.
[33] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proc. 21th USENIX Security Symp.* 2012, Art. no. 27.
[34] P. von Styp-Rekowsky, S. Gerling, M. Backes, and C. Hammer, "Idea: Callee-site rewriting of sealed system libraries," in *Proc. 5th Int. Symp. Eng. Secure Softw. Syst., ESSoS* 2013, pp. 33–41.

Wei You received the PhD degree in computer science from School of Information, Renmin University of China. He is currently working as a post doctor at Department of Computer Science, Purdue University. His research interests focus on program analysis, mobile security and Web security.

Bin Liang received the PhD degree in computer science from Institute of Software, Chinese Academy of Sciences. He is currently a professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection and Web security.

Wenchang Shi received the PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences. He is currently a professor in the School of Information, Renmin University of China. His research interests include focus on information security, trusted computing, cloud computing, computer forensics, and operating systems.

Peng Wang received the MS degree in computer science from School of Information, Renmin University of China. He is currently working towards the PhD degree in Computer Science at School of Informatics and Computing, Indiana University Bloomington. His research interests focus on mobile security, malware analysis and Web security.

Xiangyu Zhang received the PhD degree in computer science from the Department of Computer Science, the University of Arizona. He is currently a professor in the Department of Computer Science, Purdue University. His research interests include lie in techniques on automatic debugging, dynamic program analysis, software reliability, and security. He is a member of the IEEE and the IEEE Computer Society.