Reducing Security Risks of Suspicious Data and Codes Through a Novel Dynamic Defense Model

Zezhi Wu, Xingyuan Chen, Zhi Yang, and Xuehui Du

***Abstract*— A remarkable characteristic of the modern operating system is open, which means that we can download data or execute codes from any Internet sources whether they are trusted or not. Therefore, there is a contradict situation when we want to use these data or codes as well as keep the system secured. Despite decades of studies and experiences on this problem, it is still assumed as a great challenge. This paper presents a novel dynamic defense model (DDM) to reduce security risks brought by these suspicious data or codes for the open operating systems. DDM is a high-level security defense abstraction with four key components: dynamic label marking, dynamic label tracking, dynamic label modulating, and run-time controlling. With these components, DDM achieves the full, dynamic, and real-time security protection in the whole life cycle of the operating system. We also practically implemented a prototype system named DDDroid on Android. We constructed a mixed experimental dataset with 30 malware samples and 970 benign applications to test the defense effects of DDDroid. DDDroid detects 97% of the malware samples that have malicious actions and blocks these actions with a negligible false positive on legal actions. We also demonstrated that DDDroid is an effective system, which prevents sensitive data from being leaked by suspicious applications deliberately or by users unintentionally through some sample experiments. What is more, with extensive evaluations, DDDroid is proved to be a system with low-performance overhead and limited memory consumption.**

***Index Terms*— Dynamic defense model, dynamic taint tracking, information flow control, behavior-based malware analysis, android.**

I. Introduction

OPEN is a remarkable characteristic of the modern operating system (OS). These systems are allowed to exchange data and codes with the outside world. Meanwhile, due to the plentiful resources on the Internet, most of the users are willing to download data or execute codes from Internet sources which are not fully trusted. In mobile OS such as Android, users may install applications directly through third-party market. Users have to trust that the application does not contain any malicious code implicitly so as to enjoy the convenience and entertainments brought by them. While these suspicious data or codes may bring the security risks in a range from leaking sensitive data to destroying the whole operating system. As a consequence, it comes a contradict situation as we want to use these data or codes and keep the system security at the same time. Building a system that meets both the open and security requirements is a great challenge.

To guarantee the security of the OS, a lot of security mechanisms were invented. These works include intrusion detection [1], access control [2], information flow control [3], [4], malware scanners, virtual machine execution [5], moving target defense [6] and so on. OS primitives and access control mechanisms can protect sensitive resources from being unauthorized assess, but these mechanisms are vulnerable to privilege promotion attacks and can not provide an end-to-end security solution [7]. Information flow control technology can guarantee end-to-end security. Most of these works (such as Asbestos [3], Flume [4]) provide a model with application-defined security policies for individual applications to allocate labels and declassify labels, but provide little contribution for a system administrator to implement system-wide security policies. Malware scanners and similar techniques are used to detect malicious codes. However, these techniques rely on existing knowledge and can not deal with new malicious codes properly. Moving target defense focuses on how to prevent the attackers from intruding into the system, but it provides little help to evaluate the codes which are downloaded by users. At the same time, most of existing security mechanisms belong to the type of boundary control mechanism. Once the malicious data or codes have passed the boundary controlled by the mechanism, there is no more security measure provided by the mechanism to protect the system. Therefore, there is an urgent need to build an abstract model which provides a set of simple yet powerful primitives to reduce the security risks brought by the untrusted data and codes.

We address this problem by presenting a novel dynamic defense model (DDM). DDM is a high-level security defense abstraction with four key minds: dynamic label marking, dynamic label tracking, dynamic label modulating and run-time controlling. Firstly, for all the suspicious data and codes, we can not fully trust them. So we should mark them with a distinct label to distinguish them from trusted data or codes. For all the sensitive data, we should also mark them

Manuscript received November 9, 2017; revised April 18, 2018 and November 4, 2018; accepted February 20, 2019. Date of publication February 26, 2019; date of current version June 5, 2019. This work was supported in part by the National High Technology Research and Development Program under Grant 2018YFB0803603). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Lorenzo Cavallaro. *(Corresponding author: Xingyuan Chen.)*

Z. Wu and X. Chen are with the Institute of Zhengzhou Information Science and Technology, Zhengzhou 450001, China, and also with the State Key Laboratory of Cryptology, Beijing 100084, China (e-mail: 1141208772@qq.com; chxy302@vip.sina.com).

Z. Yang and X. Du are with the Institute of Zhengzhou Information Science and Technology, Zhengzhou 450001, China (e-mail: zynoah@163.com; dxh37139@sina.com).

This paper has supplementary downloadable material at http://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/TIFS.2019.2901798

with a distinct label in case of arbitrary usage or leakage. Secondly, with the running of the OS, suspicious data and sensitive data will be used and propagated in the system, and child process will be created by the suspicious processes. We should keep an eye on where these data go and how much data was infected, and how these suspicious processes affect the system via dynamic label tracking. Thirdly, with the observation on the real-time executing of the suspicious processes, the initial label may be inadequate to present the current security station of the suspicious processes. We should modulate the label dynamically according to the recent behavior of the suspicious processes. Lastly, for all the data and codes, we should not let them propagate or run arbitrarily in the system without any restriction. Malicious actions should be blocked according to the label-related security policies to prevent any potential damage to the system.

We implemented a prototype system named DDDroid on Android, which is the most popular and vulnerable operating system nowadays [8], [9]. DDDroid provides a robust dynamic defense method by integrating the existing security work (such as dynamic taint tracking [10], behavior-based malware analysis [11], [12], in-device intrusion detection [1], access control [2], decentralized information flow control [13], [14], risk analysis [15], [16]) on Android together organically. The key design goals of DDDroid are efficiency, convenience and compatibility. For efficiency, We presented some optimization methods to accelerate the label tracking system. For convenience, several applications (DDPolicy, DDFile and DDNotify) are provided for convenient usage for common users. For compatibility, any suspicious third-party application can successfully run in DDDroid without any modification or preview analysis. We random choose 30 malware samples (which are totally unknown to us) and 970 benign samples to validate the correctness of the implementation of DDDroid. The results show that DDDroid detects and blocks 97% of the malicious actions performed by the malwares with a negligible false alarm on legal actions performed by benign applications. We also demonstrated that DDDroid is an effective system that prevents the sensitive data from being leaked by suspicious applications deliberately or by users unintentionally through some sample experiments. What's more, with the extensive evaluations, DDDroid is proved to be a system with low performance overhead and limited memory consumption.

The main contributions of this paper include: 1. A novel dynamic defense model is proposed. Through dynamic label marking, tracking, modulating and controlling, the security risks of suspicious data and codes can be reduced theoretically. 2. A prototype named DDDroid is implemented on Android. DDDroid provides a comprehensive method to address the security problems for Android. Experiment results demonstrated that DDDroid provided a robust and effect dynamic defense mechanism for Android.

II. Related Works

A. Malware Analysis and Defense

Static analysis is an important way to reduce the security risks of untrusted codes. MCC (model-carrying code) [17] provides a tool for users to make decisions about the security risks they are willing to tolerate so as to enjoy the functionality provided by untrusted code. Static binary translation is used to translate the untrusted codes to trusted codes according to user-defined policies. Meanwhile, a mount of static malware analysis methods [18] based on machine learning are presented to distinguish the malicious codes, but these methods depend on observation of sufficient malware samples. What's more, static analysis can not deal with code transformation and obfuscation technologies [19] effectively. Dynamic analysis by monitoring the execution of a running program can overcome the limitations of static analysis, but these methods can not deal with the anti-virtualization and anti-debugging [20] technologies used by modern malware.

Executing untrusted codes securely in real-time has been a challenging task of the operating system. Sandbox is a common approach for defending against malicious codes. The resource accesses made by untrusted codes are suitably restricted to ensure security. An alternative approach to sandbox is isolated execution. The execution of the untrusted codes are isolated [21] from the trusted codes physically or logically. Safe-loading [22] provided a trusted runtime environment which enables the safe execution of the untrusted program by using a secure loader and a user-space sandbox. Jiang and Akhter [23] proposed three protection tiers to execute and evaluate the untrusted code on a Linux-based web server. However, Sandbox and isolation provide little support for enforcing highly restrictive security policies. Above works focus on implementation a specific system that ties to a particular language or runtime environment. They also lack a security model to abstract the different kinds of security policies. Despite decades of researches and experiences on this problem, it is still a great challenge to build a secure operating system to fulfill this security requirement.

B. Taint Analysis and Information Flow Control

Anomalous taint detection is the most related technique to our work, which combines dynamic taint tracking (DTT) and anomaly detection based approaches. As we know, anomaly detection technique suffers of false positives, and DTT is a prevalent technique in malware analysis fields due to its high accuracy. With the help of the information provided by DTT, the drawbacks of anomaly-based technique can be mitigated. Inspired by anomalous taint detection, we proposed a security defense model to reduce the potential risks of suspicious data and codes. Most of DTT works are suffering heavy overhead. The methods explored by previous researches to improve the efficiency of DTT systems include: On-demand taint [24], paralleled and decoupled taint [25], [26], share taint [27], lazy taint [28], tagMap collapse [29], fast-path [30], merged check [30], fast-switch [30] and statical taint [31]. However, Some of these works require availability to source codes or additional hardware resources such as CPU cores (the frequent synchronized communication between original execution CPU and taint tracking execution CPU leads huge expenses and delay of the original execution) and DBI platform. Some of these works only perform well under certain conditions

(such as when the tainted data is seldom accessed). We focus on the characteristics of colorful taint tracking and streamline taint tracking code with various code optimization techniques such as eliminating, replacing and moving for in-lined DTT. Our methodology is compatible with android framework perfectly, and do not require availability to application source codes, DBI platform or additional resources such as CPU cores.

Information flow control is an outstanding technique to protect sensitive data from being leaked. Classic information flow control was initially applied in military systems in order to protect information confidentiality and integrity. Traditional information flow controls models, such as BLP and BIBA, are too rigid to use in majority of the cases. Myers and Liskov [7] first presented a Decentralized Label Model (DLM) in Jif [32] to control information flow at compile time (statically) in decentralized authority systems. Jif labels variables and objects at a fine granularity, but it requires an entirely new programming language which is incompatible with legacy software designs. Asbestos [3], HiStar [33], Flume [4] and Aeolus [34] are different kinds of DIFC OSs. They enforce information flow control through labels and tags to protect OS resources like files and sockets at runtime, but they only provide process-level information flow tracking which is coarse grained. Dstar [35] and Fabric [36] extended DIFC from the single OS to the network environment. Laminar [37] implemented a DIFC system by using a hybrid of JVM and OS mechanisms to protect variables and OS resources at runtime. DIFC systems provide a richer model with application-defined security policies for individual applications to allocate labels and declassify labels, but provide little contribution for a system administrator to implement system-wide security policies.

C. Android Security

There are a lot of works on Android security [9], [38]. Most related to our work are dynamic taint tracking, behavior-based malware analysis, in-device intrusion detection, access control, decentralized information flow control, privacy preservation and quantitative risk analysis. We just introduce the most classic and the latest works in this paper. TaintDroid [10] provided a runtime taint tracking framework to avoid stealing of sensitive data. However, TaintDroid provided little contribution on enforcement of information flow control policies for users. Patronus [1] is a host-based intrusion prevention system, which can prevent intrusion and detect malware dynamically. However, Patronus provided little contribution for a user to specify privacy policies. Monet [11], MADAM [12], DroidScribe [39], and CopperDroid [40] are different kinds of malware analysis works based on run-time behavior. Monet [11] provided an user-oriented behavior-based malware variants detection method and MADAM [12] provided a multi-level and behavior-based malware detection method. DroidScribe [39] used machine learning to automatically classify Android malware samples into families. CopperDroid [40] presented a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors. However, CopperDroid relies on QEMU and it can not be deployed on Android devices directly. We make a combination of these behavior-based works and dynamic taint tracking works on Android to achieve better accuracy on malware detection. SEAndroid [2] provided a flexible MAC to Android based on the LSM module of the Linux kernel. However, SEAndroid does not take the high-level Android-specific behaviors into consideration. DroidNet [41] provided a new permission control framework through crowdsourcing for Android users. However, permission mechanism can not guarantee the end-to-end privacy security. DroidRisk [16], MAETROID [15], DroidAnalytics [42] and DroidSieve [43] are different kinds of malware detection works based on static analysis. DroidRisk [16] proposed a framework to quantitative security risk assessment of both Android permissions and applications based on permission request patterns. However, DroidRisk is simply based on the required permissions, which is not very precise. MAETROID [15] proposed a multi-criteria analysis method to evaluate the trustworthiness of an application. DroidAnalytics [42] proposed a signature based malware analysis method. However, signature and static feature based analysis can not deal with transformation and obfuscation effectively. DroidSieve [43] exploited obfuscation-invariant features to detect obfuscated malware. However, the injection of non-Java code, network activity, and the modification of objects at run-time are outside the scope of static analysis as they are only visible during execution. Jia *et al.* [13] proposed a DIFC-style enforcement system that permits applications to specify security policy via labels applied to the components on Android. Weir [14] is also a DIFC system on Android that allows applications control the export of their data to the network. However, DIFC adapts to the distributed systems well, but not for the centralized authorization systems where an administrator manages all the privileges. We try to take advantages of these existing works and provide a comprehensive method to address the security problem for Android.

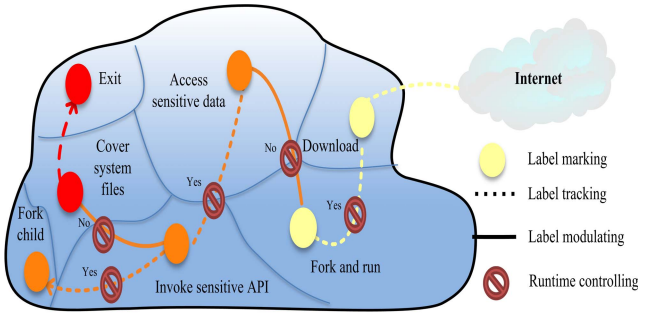


Fig. 1. An example to show how DDM protects OS from being damaged by suspicious applications.

III. Dynamic Defense Model

As we mentioned before, the four key components of DDM are dynamic label marking, dynamic label tracking, dynamic label modulating and run-time controlling.

Figure 1 shows how DDM protects OS from being damaged by suspicious programs from Internet. When a suspicious program is downloaded, DDM marks it with a *risk* label and

sets an initial value to the *risk* label. When a process is forked by executing the program, the initial value of the *risk* label of the program is set to the value of the *risk* label of the process in OS. When the process trying to access sensitive data, which is not allowed by the label-related policies, DDM blocks this action and then up-modulates the value of the *risk* label of the process. When the process trying to invoke sensitive API, which is allowed by the label-related policies, DDM then permits this action. When the process trying to fork a child process, which is allowed by the label-related policies, DDM then permits this action and sets the value of the *risk* label of current process to the value of the *risk* label of its child. Therefore, the value of the *risk* label is tracked and modulated based on the behavior of the process in the OS at runtime. Once the value of the *risk* label of the process is bigger than the threshold, DDM will kill this process.

A. Dynamic Label Marking

Label is a common concept that has been widely used in computer security fields. Generally, DDM provides three kinds of labels as:

$$label \triangleq risk \parallel cap \parallel sens$$

For all the suspicious data and codes that come from the Internet sources, we should mark them with some distinct labels to distinguish them from the trusted data or codes as soon as they come into the system. The *risk* label is designed to represent how big the risks of the suspicious data or codes are. The *cap_s* label is designed to represent the basic or least capabilities (permissions or privileges) that the suspicious code should be given in order to fulfill their function usage. For all the sensitive data or resources that need to be protected in the system, such as system files, directories, processes, user data, APIs, kernel data structures and other limited resources, we should also mark them with some distinct labels in case of arbitrary usage or leakage. The *sens* label is designed to represent how important the sensitive data or resources are. The *cap_o* label is designed to represent the required capabilities to access the sensitive data or resources.

Another key issue is how to set the initial value of these labels. For the *risk* label, the initial value of the *risk* label could be a default value simply (supposed there are four security level as low, medium, high, and very high, a default value of the *risk* label could be medium) or depends on the trusted-level of the Internet sources (the initial value of the *risk* label of application that from famous companies with a signature is lower than that from unknown individuals). For the *cap_s* label, the value could be empty or be set by the user manually. For the *cap_o* label of sensitive resources, the initial value could be the values that have already been set by other security mechanisms (such as DTE in Linux or permissions in Android). For the *cap_o* label of sensitive data, the initial value could be set depends on the well-defined interfaces. More precisely, the initial value of these labels can be quantitatively calculated by specific algorithms which have been well studied before. For example, the initial value of *risk* label can be quantitatively calculated based on existing work quantitative malware analysis [16] and quantitative information flow [44]. The initial value of the *sens* label can be quantitatively calculated based on property assessment [15].

What's more, the labels of DDM are not restricted to a fixed format or implication. They are freestyle and expandable according to the security requirements. For example, the labels used in BLP or RBAC (Role Based Access Control) model could be applied to DDM.

B. Dynamic Label Tracking

With the running of the operating system, child processes will be created by suspicious program, and suspicious data and sensitive data will be used and propagated in the system. We should keep an eye on how these suspicious program affect the system, and where these data go and how much data was infected through dynamic label tracking.

Label tracking is a common technique that has been implemented at different levels of the system, such as language-level [32](variable or object), OS-level (process, file or communication data structure) [4], hardware-level (register or memory) [45], database-level (tuple or item) [46] and network-level (packet) [35]. By analyzing these works, we capture the high-level characteristic of label tracking and define our dynamic label tracking rules based on interference. Typically, non-interference is used as an abstracted security property of an information system, which means that one security area should not interfere another security area except for the allowance of a specified security policy. We define the high level rule of our label tracking as:

$$A \sim B \Longrightarrow B.label = f(A.label, B.label)$$

The meaning of this rule is that if a security entity A interfered ($\sim$) another security entity B, then the value of the label of B should be updated to a value that calculated by function $f(A.label, B.label)$.

For the *risk* label, this rule can be specified as:

$$A \sim B \Longrightarrow B.risk = \max(A.risk, B.risk)$$

For example, if a process creates a child process, then the value of *risk* label of the child process is set to the value of the *risk* label of the parent process. If suspicious string B is constructed by a combination of string A and B, then the value of *risk* label of string B is updated to the maximum value between string A and B.

For the *sens* label, it is similarly to the *risk* label. For the *cap_o* label, this rule can be specified as:

$$A \sim B \Longrightarrow B.cap\_o = (A.cap\_o \bigcup B.cap\_o)$$

$\bigcup$ refers to the set union. For example, If sensitive string B is constructed by a combination of string A (with a label *location*) and B (with a label *contacts*), then the value of the *cap_o* label of string B is updated to an union value of string A and string B ({*location*,*contacts*}).

C. Dynamic Label Modulating

If a suspicious program does a lot of actions that go against the security policy, the initial value of the *risk* label may be inadequate to present the security state of the current program. Similarly, the data that computed upon large of insensitive data could be not insensitive data any more [47]. We should modulate the label of the suspicious data and codes to the fitness value dynamically.

DDM modulates the label in three ways: automatic incremental modulation (implicitly), automatic decremental modulation, and evaluative modulation (explicitly). Automatic incremental modulation can be understood as: only monotonically increasing of the value of the specify label is allowed. For example, the value of the *risk* label of a process increases automatically while it running in the system. The value of the *sens* label of data increases automatically while it propagating in the system. Automatic decremental modulation can be understood as: only monotonically decreasing of the value of the specify label is allowed. For example, the value of the *cap_s* label of a process decreases automatically while it running in the system. For the consideration of security, the label of DDM can not belong to both the set of automatical incremental modulation and automatical decremental modulation. For example, the value of the *risk* label can only be upward modulated and can not be downward modulated automatically. Similarly restrictions are commonly used in RBAC, which is called separation of duties.

The rule for automatic incremental modulation of the *risk* label can be descried as:

$$risk_{new} = risk_{old} + f(action_{ill})$$

$risk_{new}$ refers to the new value of the *risk* label after the illegal action (does not satisfy the security policy) $action_{ill}$, and $risk_{old}$ refers to the old value of the *risk* label before the illegal action $action_{ill}$, and function f is used to map an illegal action to a *risk* value. For example, a process with a *risk* value 3 and did an illegal action that $f(a) = 1$, then the *risk* value of this process is updated to 4.

Similarly, the rule for automatic incremental modulation of *sens* and *cap_o* label can be descried as:

$$sens_{new} = sens_{old} + f(action_{sen})$$
$$cap\_o_{new} = cap\_o_{old} + f(action_{sen})$$

The rule for automatic decremental modulation of *cap_s* label can be descried as:

$$cap\_s_{new} = cap\_s_{old} - f(action_{ill})$$

cap_s_{new} refers to the new value of the *cap_s* label after the illegal action $action_{ill}$, and cap_s_{old} refers to the old value of the *cap_s* label before the illegal action $action_{ill}$, and function f is used to map an illegal action to a *cap_s* value. For example, a process with a *cap_s* value $\{location, contacts\}$ tries to send the contacts information to unknown network address, then we drop the label of $\{contacts\}$ ($f(a) = \{contacts\}$) in its capability label. The value of the *cap_s* of this process is updated to $\{location\}$.

The rule for evaluative modulation can be descried as:

$$label_{new} = f(actions)$$

Evaluative modulation can be understood as: the value of specify label can be assigned to a new value, which does not depend on the old value, through evaluation function f directly. Evaluative modulation supports both incremental and decremental modulation manually. This is very similar to the conditional (explicit) declassification or endorsement in DIFC models. For example, the value of the *risk* label of a process can be decided upon its recent behaviors in the system. For the consideration of security, evaluative modulating must be done explicitly or manually by the administrator or user.

D. Runtime Controlling

For all the data and codes, we can not let them propagate or run arbitrarily in the system without any restriction. We should block malicious actions according to security policies to prevent any potential damage to the system or the leakage of sensitive data.

Whether an action is illegal depends on runtime controlling rules, and the runtime controlling rules are all related to the labels used in DDM. Therefore, dynamic label marking, dynamic label modulating and runtime controlling are three interdependent processes in DDM. With the benefits of these three processes, each label has rich contextual information of the data and codes in the system. It is scientific to control the future behavior of the data and codes based on these labels. The high-level of runtime controlling rules can be described as:

$$action\ is\ permitted \Longleftrightarrow f(labels) == yes$$

Function f is used to map the labels to the decisions *yes* or *no*.

For the *risk* label, this rule can be specified as:

$$running\ is\ permitted \Longleftrightarrow risk < threshold_{kill}$$

This rule means that a process can keep running if and only if the value of its *risk* label is less than the threshold of killing a process.

For the *risk* label and *sens* label, this rule can be specified as:

$$access\ is\ permitted \Longleftrightarrow risk + sens < threshold_{sys}$$

This rule means that a process with a *risk* label can access the sensitive data or resources with a *sens* label if and only if the value of the *risk* plus *sens* is less than the threshold of the system.

For the *cap_s* and the *cap_o* label, this rule can be specified as:

$$access\ is\ permitted \Longleftrightarrow cap\_o \subseteq cap\_s$$

This rule means that a process with a *cap_s* label can access the sensitive data or resources with a *cap_o* label if and only if the value of *cap_o* is a subset of *cap_s*. For example, data with *cap_o*=$\{location\}$ can only be accessed by the application who has $\{location\}$ in its *cap_s* label.

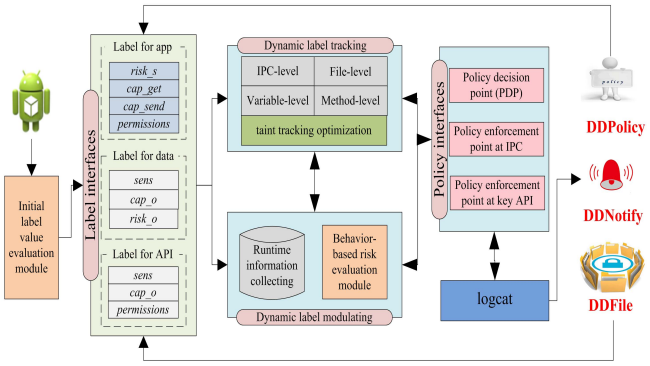


Fig. 2. The architecture of DDDroid.

IV. Design and Implementation

In this section, we introduce our design and implementation of DDDroid on Android. Android provides two important security mechanisms for users. one is sandbox mechanism, each application runs isolate in its VM instance with an unique UNIX identity. The other one is permission mechanism, application only can use the permissions (explicitly declared in their manifest files) accepted by users. DDDroid takes advantages of these security mechanisms and adds some new security mechanisms at the application framework layer and VM layer of Android. Figure 2 shows the architecture of DDDroid. DDDroid has the same four key components as DDM.

A. Dynamic Label Marking

1) Labels for Application: Besides permissions, we add three labels for each application in Android. They are $risk\_s$, *cap_get* and *cap_send*. $risk\_s$ is used to represent the potential risk value of a suspicious application. *cap_get* is used to represent the capabilities of a suspicious application on what kind of sensitive data it is allowed to access. *cap_send* is used to represent the capabilities of a suspicious application on what kind of sensitive data it is allowed to send out from the system. The labels are classified into get and send in our system instead of secrecy and integrity in the DIFC systems. This offers a better description of information flow characteristics for a common user to understand. All a user need to do is define a policy on what kinds of applications can get or send what kinds of private data based on his security requirements. $risk\_s$, *cap_get* and *cap_send* are each independently implemented as an integer number in our system.

For the initial value of $risk\_s$ label, we analyze the permissions that declared by the application in its AndroidManifest.xml and perform a preliminary risk assessment to compute a score (which is represented by an integer number that ranging over the interval [0,15]) when it is being installed. The score is proportional to the dangerousness (privacy leakage, integrity violation, malicious charge, and vulnerability) of the requested permissions and the number of the requested permissions. To construct our assessment algorithm efficient, we simply give each permission with a risk score ranging from 0 to 1 on its dangerousness, which benefit from the existing works DroidRisk [16] and MAETROID [15]. Table I shows the partial list of Android permissions and associated risk value that pre-defined by us. Take the permission android.permission.CALL_PHONE as an example. We have assigned a score of 0.6 to the corresponding privacy leakage, since an application may perform phone call background to leak privacy. We have assigned a score of 0.2 to the system integrity violation, since the phone drains the battery faster during a call. We have assigned a score of 1 to the malicious charge, since an application may call premium-rate phone numbers. Then we calculate the risk score by pulsing all the scores of required permissions together. We restrict the biggest initial value of the $risk\_s$ label is 15. If the calculated score is bigger than 15, we then set 15 to the initial value of *risk_s*. This is due to our assessment algorithm is simply based on the required permissions, which is not very precise. Some of the benign applications also applied lots of sensitive permissions. If we dont restrict the biggest initial value of the *risk_s* label, some of the benign application cannot even run in our system firstly or cannot access some sensitive resources. Value 15 is chosen based on our runtime controlling rules which we will explain on sub-section D. For the initial value of *cap_get* and *cap_send* label, they are initialed according to the type of application and the permissions being allowed by the user when the application is being installed. For example, if an application belongs to the type of CHAT, and permissions READ_CONTACTS is granted by the user, we then automatically set the *cap_get* label with value 0x00002 (LABEL_CONTACTS: 0x00002). We also provide an application named DDPolicy for users to set their desire value.

TABLE I
Partial List of Android Permissions and Associated Risk Value

Permission	Privacy leakage	Integrity violation	Malicious charge	Vulnerability
FINE_LOCATION	0.8	0	0	0
NETWORK_STATE	0.2	0	0.4	0.2
BATTERY_STATS	0	0.2	0	0.4
BROADCAST_STICKY	0	0.2	0	0.6
CALL_PHONE	0.6	0.2	1	0
CAMERA	0.8	0.6	0	0
WRITE_CALENDAR	0.8	0.2	0	0
WRITE_CONTACTS	0.6	0.6	0	0
RECEIVE_MMS	1	0	0.8	0
RECEIVE_SMS	1	0	0.8	0
RECORD_AUDIO	0.8	0.6	0	0
REORDER_TASKS	0.4	0.2	0.4	0.2
CHANGE_CONFIG	0	0.4	0	0.6
CLEAR_APP_CACHE	0	0.2	0	0.2

When a suspicious application is being installed on the phone, we intercept and hijack the installation event and analyze this application to assess its initial *risk* value, and then pop a new dialog box to show the initial risk value. A user can choose whether to continue the install progress depends on this value. If the user chooses to install, then we pop another dialog box to require the initial value of the *cap_get* and *cap_send* label. The user could set these values depends on what kinds of functions he wants to use of the suspicious application. For example, if the suspicious application is MAP, then the user could set the *cap_get* and *cap_send* with a value 0x00010 (which is used to mark the GPS location data in DDDroid)

Label interfaces for application:
- void *SetRisk*(*risk_s*): set *risk_s* label for current app.
- void *SetCap_get*(*cap_get*): set *cap_get* label for current app.
- void *SetCap_send*(*cap_send*): set *cap_send* label for current app.
- int *getRisk*(): get *risk* label for current app.
- int *getCap_get*(): get *cap_get* label for current app.
- int *getCap_send*(): get *cap_send* label for current app.

Label interfaces for suspicious data:
- void *setString*(*string*, *risk_o*): set *risk_o* label for string.
- char *setChar*(*char*, *risk_o*): set *risk_o* label for char.
- int *setInt*(*int*, *risk_o*): set *risk_o* label for int.
- int *getString*(*string*): get *risk_o* label for string.
- int *getChar*(*char*): get *risk_o* label for char.
- int *getInt*(*int*): get *risk_o* label for int.

Label interfaces for sensitive data:
- void *setStringC*(*string*, *cap_o*): set *cap_o* label for string.
- void *setStringS*(*string*, *sens*): set *sens* label for string.
- int *setFileC*(*fd*, *cap_o*): set *cap_o* label for file.
- int *setFileS*(*fd*, *sens*): set *sens* label for file.
- int *getStringC*(*string*): get *cap_o* label for string.
- int *getStringS*(*string*): get *sens* label for string.
- int *getFileC*(*fd*): get *cap_o* label for file.
- int *getFileS*(*fd*): get *sens* label for file.

Fig. 3. Partial of the label interfaces that provided by DDDroid.

so as to use the map function provided by MAP. At last, we store these values of the labels into a policy file named LabelPolicy.xml for eternal storage and later usage.

When the system initialing the package manager service, we parse the policy file and store all the policies into HashMap. When the system calling the function *scanPackageLI*, we check whether the HashMap contains the key of the name of current application. If contains, we fill the data structures of the current application in *pkg.applicationinfo* according to the values that have been stored in HashMap for fast access. Otherwise, we set value 15 to the *risk* label and value 0 to the *cap_get* and *cap_send* label as default. As we know, Zygote is the parent process for all applications in Android. When a suspicious application is started by invoking *Process.Start* in *ActivityManagerService*, we add our label parameters to function *Process.Start*, and pass the label parameters through multi-function invoking such as *startViaZygote*, *nativeforkAndSpecialize* and *forkAndSpecializeCommon*. At last, we set the label for this application through the label interfaces that designed for application, which is shown in Figure 3. We define these interfaces in *LabelInterf.java* and implement them through JNI in *LabelInterf.cpp*.

2) Labels for Suspicious or Sensitive Data: We add *risk_o* for the suspicious data to represent the potential risk value of the suspicious data. For the initial value of *risk_o* label, we perform a preliminary risk assessment to compute a score (which is represented by an integer number that range over the interval [0, 31]). The score is proportional to the dangerousness of the Internet sources and the *risk_s* value of the receiver application. We define five risk levels to describe the dangerousness of the Internet sources. They are trusted (*risk* value 5), moderate trusted (*risk* value 10), low trusted (*risk* value 15) and suspicious (*risk* value 20) and malicious (*risk* value 31). For some of the famous Internet sources such as BAIDU and QQ, we define them as trusted. For some of the Internet sources that do not list in our white list, we define all of them as suspicious. For some of the Internet sources that list in our black list, we define all of them as malicious. We also assign the data received from secure protocols such as SSL with an additional risk value of 0 and

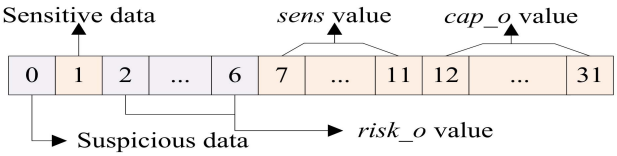


Fig. 4. The meaning of each bit of the label (unsigned int) for data.

the data received from other protocols with an additional risk value of 10. To construct our assessment algorithm efficient, we simply calculate the initial *risk* value through pulsing the risk values of Internet sources and the *risk_s* value of the receiver application. For example, application WECHAT (*risk_s* = 6) received data from *qq.com* (*risk* = 5) with protocol SSL (*risk* = 0), we then set the data with an initial *risk_o* value of 11 (6+5). If the protocol is HTTP, we then set the data with an initial *risk_o* value of 21 (6+5+10). If the total value is bigger than 31, then set 31 to the value of the *risk_o* label (this is because we use five bits to represent this value in an unsigned integer number in Figure 4).

We also add the *sens* and *cap_o* for all the sensitive data. *sens* is used to represent the importance of the sensitive data and *cap_o* is used to represent what kinds of capabilities of an application are needed to access the data or send the data out from the system. For the initial value of the *sens* label, we simply give each kind of sensitive data with a value ranging from 0 to 5. Similarly to *risk_s*, the value is proportional to the sensitiveness of the resources. For the initial value of the *cap_o* label, it depends on what kinds of sources or well-defined interfaces that the data come from. For example, data received from the GPS interface is marked with a value of 0x00010.

We use an unsigned integer to implement these labels for suspicious or sensitive data. Figure 4 shows the meaning of each bit or several bits in the unsigned integer. The first two bits are used to represent whether the data is suspicious data or sensitive data. 3-7th bits are used to represent the *risk* value of suspicious data. 8-12th bits are used to represent the *sens* value of the sensitive data, and 13-31th bits are used to represent its *cap_o* value. We provide different kinds of label marking interfaces for different kinds of data, such as int, long, float, double, string, array and so on. Figure 3 shows the partial of them. For example, We mark contacts data with a label 0x00002 at contentresolve.java through interface *setStringC(string, cap_o)*. We also add the *sens* label for the sensitive API, which is similar to the *sens* label for sensitive data.

B. Dynamic Label Tracking

We use TaintDroid to track our label at four different levels. They are variable-level (data flow of each VM instruction), message-level (IPC between applications), method-level (invocation between Java and native codes) and file-level.

As we know, TaintDroid is one of the outstanding DTT systems that provide fine-grained taint tracking on Android. TaintDroid used a 32-bit bit-vector for each local variable to encode the taint tag, which we call it as colorful taint tracking. TaintDroid modified the interpreter and the just-in-time

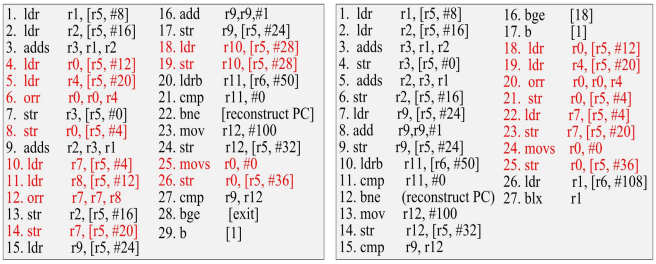


Fig. 5. Taint tracking implementation and optimization on the example codes at variable-level in DDDroid.

compiler of the DVM. To implement taint tracking for the JIT compiler of DVM, take the DVM instruction [binop vAA, vBB, vCC] as example, there are three steps were added by TaintDroid. First, it loaded the taint value of vBB and vCC from the stack to physical register through function loadTaintDirect(cUnit, vBB, taint) and loadTaintDirect(cUnit, vCC, taint2). Then, it calculated the new taint value through function opRegRegReg(cUnit, kOpOr, taint, taint, taint2). Last, it stored the taint value to stack through function storeTaintDirect(cUnit, vAA, taint). If we naively insert additional taint tracking codes for each VM instruction independently according to the template, it would bring substantial runtime overhead. Especially for loops, these additional taint tracking codes will be executed many times. Take following codes as example:

```
                          : goto_0
for(i=0; i<100; i++)        . const/16 v4, 0x64
{                           . if-ge v3, v4,:cond_0
  a=b+c;                  : cond_0
  c=a+b;                    . add-int v0, v1, v2
}                           . add-int v2, v0, v1
                            . add-int/lit8 v3, v3, 0x1
                            goto: goto_0
```

For DVM instruction *add-int v0, v1, v2*, to realize the taint tracking, the compiler first loads the taint tags of *v1* and *v2* from memory locations (stack point [*r5*, ♯12] and [*r5*, ♯20]) into physical registers *r0* and *r4*. Then it adds the native taint tracking instruction *orr r0,r0,r4* to calculate the new taint value of *v0*. At last, it stores the new taint value from physical register *r0* into memory location (stack point [*r5*, ♯4]). First column of Figure 5 shows the taint tracking implementation of the example codes. The native instructions in red color are added for taint tracking by TaintDroid. We find that a lot of taint tag propagation operations contained in this hot traces are redundant. We also find that the values of some variables are changeable but their taint values are invariant in this loop.

To improve the efficiency of DDDroid, We deployed some light-weight taint propagation optimization methods (such as eliminating, replacing and moving) on hot traces with the help of JIT compiler. Based on the observation that, for most android applications, the majority of executing time is cost on minority part of the codes (only 2% of system_server application are identified to be hot traces by interprets with profiling). So it is unnecessary to optimize every piece of tracking code. Our methods include: redundant taint load elimination (RTLE), redundant taint store elimination (RTSE), redundant taint compute elimination (RTCE), taint load hoisting (TLH), taint store sinking (TSS) and loop invariant taint motion (LITM).

RTLE is used to eliminate the redundant taint load instructions. For example, considering following instructions in Figure 5: *ldr r10, [r5, #28]* and *str r10, [r5, #28]*. After loading the taint from DVM register *v7* into physical register *r10*, there is no use point of *r10* in the subsequent instructions besides instruction *str r10, [r5, #28]*, which has the same DVM register *v7*. Apparently, this kind of load instruction is unnecessary and can be eliminated. Secondly, considering instructions in Figure 5: *str r0, [r5, #4]* and *ldr r7, [r5, #4]*. Since the value of physical register *r0* is not clobbered and the value of DVM register *v0* is no updated in between these two instructions, we can reuse physical register *r0*. Generally, we can reuse the previously taint value in physical register rather than perform a new load. Case 1: there is a load instruction after a store instruction which has the same DVM register, and the physical register is not clobbered in between, and the taint value of DVM register is not updated in between. If they have the same physical register, then the later load can be eliminated. If they have different destination physical registers, then the later load can be replaced with a move. Case 2: two load instructions load the same DVM register, and the physical register is not clobbered in between, and the taint value of DVM register is not updated in between. If they have the same destination physical register, then the later load can be eliminated. If they have different destination physical registers, then the later load can be replaced with a move. RTSE is to eliminate the redundant taint store instructions so that not all computed taint values are must written back to memory location. Considering following instructions: *add-int v2, v0, v1* and *add-int v2, v3, v0*. The taint value of *v2* will be stored twice and the later taint value will overwrite the earlier taint value. We can eliminate the earlier store. Generally, two store instructions that store the taint value to the same DVM register, and the DVM register is not used in between, then the earlier store can be eliminated. RTCE is to eliminate the redundant taint calculation (orr) instructions. LITM is to move the taint tracking codes out of the loop body for simple counted loop trace. In example codes, we find that *v3* is an induction variable and its taint is loop invariant, *v4* is a constant and its taint is always *clean*. At the same time, the value of variable *v0* and *v2* are changeable but the taint value of *v0* and *v2* are invariant after the first iteration in the loop. Generally, a loop trace is considered to be a counted loop trace if it has one basic induction variable, and the loop back branch compares the basic induction variable with a constant. A loop trace is considered to be a simple loop trace if it only has one exit block and cannot throw any exceptions. We can eliminate the taint propagation instructions in the loop body and add taint propagation instructions to the exit block. We also implemented ld/st scheduling through TLH and TSS to aggressively hoist taint loads and sink taint stores

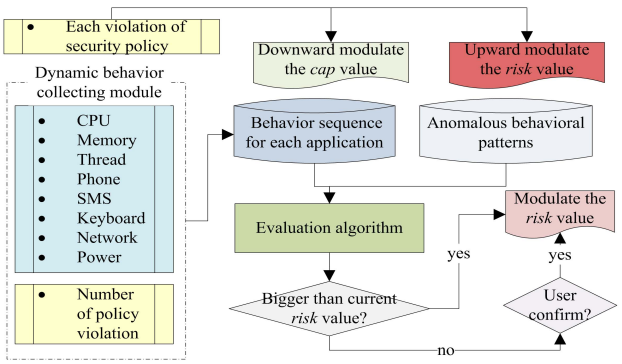


Fig. 6. The sketch process of dynamic label modulating of DDDroid.

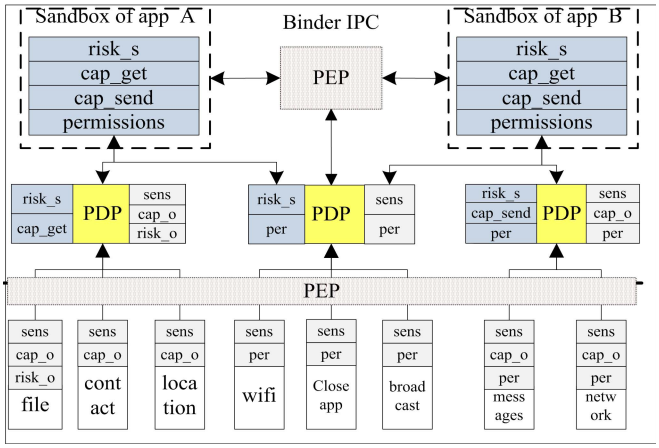


Fig. 7. The infrastructure of runtime controlling of DDDroid.

until inserted scheduling barriers or memory instructions that cannot be disambiguated.

C. Dynamic Label Modulating

We modulate the label of current application in three ways: incremental modulation automatically, decremental modulation automatically and evaluative modulation manually. Figure 6 shows the architecture of our dynamic label modulating. For incremental modulation, when a policy-violation action occurred, we then upward modulate the value of *risk* label of the current application through $risk\_s_{new} = risk\_s_{old} + sens_{actions}$.

For decremental modulation, when a policy-violation action occurred, we then downward modulate the value of *cap_get* and *cap_send* label of the current application through $cap\_get_{new} = cap\_get_{old} - v(action)$ and $cap\_send_{new} = cap\_send_{old} - v(action)$. For example, When an application tries to send the contacts information to unknown internet address which is not allowed by the policy, we then drop the capability $\{contacts\}$ in its *cap_get* label.

For evaluative modulation, first, we collect all the runtime behavior information for each application and store them in data center. Then, our evaluation module analyzes these information at fixed intervals or special point and computes a new *risk_s* value. If the new value is bigger than the previous, then we assign it to the application directly. If the new value is smaller than the previous, we then pop a dialog box to ask the user to make a decision.

The new *risk_s* value is computed based on the recent behavior and the number of policy violation of the application. $risk\_s = risk\_s_{beh} + risk\_s_{vio}$. Following are the malicious behaviors that have been reported: 1. Privilege escalation attacks to gain root access of the device. 2. Privacy leakage or personal-information theft. 3. Compromise the device to act as a Bot and remotely control it through a server by sending various commands. 4. Malicious charge through sending SMS or making phone call background. 5. Download potentially unwanted apps. 6. Denial of Service (DoS) attack when overuses already limited CPU, memory, battery and bandwidth resources and restrains the users executing normal functions. We defined following anomalous behavioral patterns for fast detection and evaluation: 1. High number (5 in a minute) of sending messages. 2. High number (5 in a minute) of sending sensitive data through network interface. 3. Sending messages at background. 4. Making phone call at background 5. Occupying too much (30%) memory. 6. Creating too many child threads. 7. Occupying too much (50%) CPU. Most of the value of these parameters are chosen based on existing works such as Monet [11], Andromaly [48] and MADAM [12], and our experiences. To construct our evaluation algorithm efficient, we simply give each kind of anomalous behavior with a corresponding weight w ranging from 3 to 8. Then we calculate the $risk\_s$ as follows:

$$risk\_s = \sum_{i=1}^{7} w_i \cdot h + \sum_{i=1}^{n} sens_n$$

h can be 1 or 0, which means that the anomalous behavior is detected or not. n is number of policy violation in a fixed interval *e-time*.

Furthermore, due to the dynamic label modulating, we believe more contextual information could be added for our runtime controlling. We can easily specify time-related or location-related policy. For example, assuming that application MS is only allowed to send messages during 8:00-22:00, DDM can achieve this goal by assigning the sending messages capability label to MS during 8:00-22:00, and dropping the sending messages capability of MS during other time via time-related dynamic label modulating.

D. Runtime Controlling

Figure 7 shows the architecture of our runtime monitoring and controlling. All the judgments are made based on the labels so that there is no need to specify a policy database. Different kinds of labels are used depending on different kinds of resource access in the system. We deploy policy enforcement point (PEP) at all the security-related points in the system. They are located at Parcel.java, cdmaSMSDispatcher.java, posix.java, OpenSSLSocketImpl.java and so on. Fox example, as we know, applications communicate via the binder mechanism, which provides transparent message passing based on parcels in Android. We enforce our policy

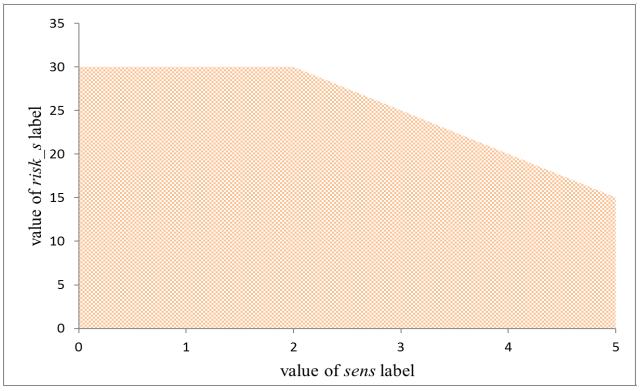


Fig. 8. The area of satisfaction values of the *risk_s* and *sens* label.

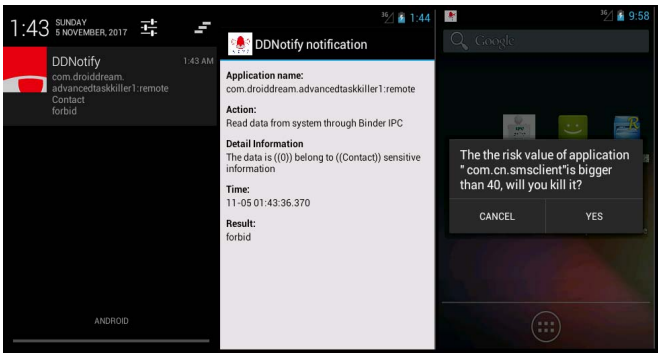


Fig. 9. The results of experiments on applications Droiddream and Smsclient.

when application writing or reading data from parcels in Parcel.java.

We deploy our policy decision point (PDP) in PDP.java. Methods defined in PDP.java take labels as input and output a decision. For *risk_s* and *sens* label, the policy is defined as:

$$risk\_s \leq 30 \quad \texttt{and} \quad risk\_s \leq 40 - 5 \times sens$$

The shadow area in Figure 8 illustrates the values of the *risk_s* and *sens* label that satisfy these rules. If the value of the *risk_s* label of an application is bigger than 30, then the application can only run in the system but cannot do any sensitive operation. If the value of the *risk_s* label of the application is smaller than 15, it can do any sensitive operation (without the consideration of other labels).

For *cap_get*, *cap_send* and *cap_o* label, the policy is defined as:

$$cap\_o \subseteq cap\_get \quad , \quad cap\_o \subseteq cap\_send$$

For example, when an application with a *cap_get* label reading data with a *cap_o* label from parcels, we use "if(*decision*(*cap_get*,*cap_o*)) then allow; else forbidden;" to enforce our policy. *decision*(*cap_get*,*cap_o*) is defined as *cap_o*==(*cap_o*&*cap_send*). When an application with a *cap_send* label sending data with a *cap_o* label to network, we use "if(*decision*(*cap_get*,*cap_o*)) then allow; else forbidden;" to enforce our policy. Therefore our PEP is very simple and efficient.

We also provide DDNotify, a stand alone application that notify users when a malicious application is detected (the value of the *risk_s* label is bigger than 40).

V. Experiments

Experiments were performed in Android emulator to demonstrate our implementation is correct and effective. We set the value of fixed interval *e-time* to 1 minute (just for experiments) and the threshold of killing a process to 40.

A. Experiment 1

In this experiment, we demonstrate that DDDroid is an effective dynamic defense system that can detect and block malicious actions at runtime with low false positive (FP). First, we construct our experiment dataset with 30 malware samples and 970 benign applications. These malware belong to different malware families from three datasets (Genome [49], Drebin [50] and DroidAnalytics [42]). Benign applications are chosen based on download number of AnZhi market. All the initial values of labels of the experiment dataset are configured automatically without manual intervention. To simulate user's actions, we use monkey tool to generate different kinds of user/system events. Then, we run each of them for ten minutes and record the malicious actions which were detected in DDDroid. DDDroid detects the 97% (true positive) of the malware samples have at least one malicious action. Figure 9 shows that DDDroid blocks these actions effectively. When an application has done too many bad jobs and its risk value is bigger than 40, DDDroid pops a dialog box to ask the user whether killing the application or not. There are 12 false positives (1.2%) are alarmed by DDDroid. some of the FPs are caused by the inadequate initial value of the *cap_get* and *risk_s* label. Some of the FPs are caused by the taint tracking explosion on String. We also analyzed the relation between the time and the number of FPs. We found that most of (99%) the FPs are alarmed in 3 minutes. To gain further insight into perceived the relation between the time and the number of FPs. We also build our system on real device. We used this device daily in practice. Few additional FPs (2 or 3 per week) was occurred.

B. Experiment 2

We demonstrate that DDDroid provides a fine-grained sensitive data protection mechanism which can prevent the sensitive data from being leaked. In this experiment, DNotify sends notification once a malicious action is detected. Following is a common security requirement. In order to ensure the functional availability, application WPS is allowed to read sensitive file and send data out via the network interface. Nevertheless, in order to ensure the privacy and security, WPS is not allowed to send the sensitive files (even a word or a sentence in the file) out via the network interface or receive contacts information from application Message. Similarly, application Message is allowed to read sensitive files or contacts data and send messages information out via the message interface, but Message is not allowed to send the sensitive file or contacts information out via the message interface. To the best

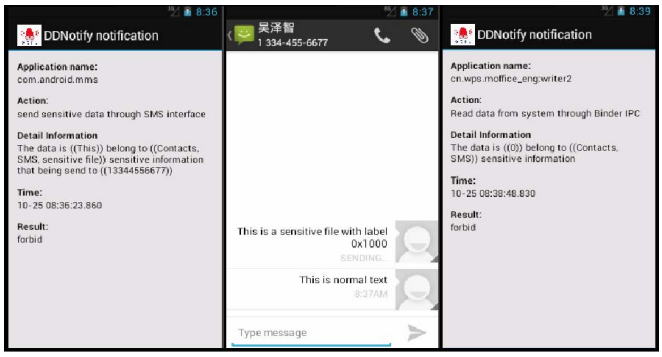


Fig. 10. The results of experiments on applications MS and WPS.

of our knowledge, no existing work can fulfill this security requirement. To fulfill this requirement, we first set the value of the *cap_o* label of file /data/sensitive.doc with 0x10000. We set the value of the *cap_get* and *cap_send* label of WPS with 0x10000 and 0x0, so that it can read the sensitive files but can not send any sensitive data out from the system (this is done by application DDPolicy and DDFile). We also set the value of *cap_get* and *cap_send* label of MS with 0x10600 and 0x600, so that it can read the sensitive files and contacts and messages, and can not send sensitive files out from the system. Then, we use WPS to open a sensitive file, which is allowed by the policy. The result is that WPS opened the file successfully. We try to copy some contacts data from MS to WPS, which is not allowed by the policy. Figure 10 shows the results that WPS is failed to get contacts data (DDNotify sends the audit information to user through notification). At last, we use MS to send contacts data via the SMS interface and copy some words from WPS to Message, which is allowed by the policy, and we try to send the copied words via the SMS interface, which is not allowed by the policy. Figure 10 shows the results that Message sends the contacts data out successfully and is failed to send sensitive file out (DDNotify sends the audit information to user through notification).

VI. Evaluation

Evaluations were performed on Android emulator with starting command: ëmulator -kernel qumu-armv7 -system system.img -ramdisk ramdisk.img -data userdata.img -partition-size 400 -memory 512. The host operation system is ubuntu 12.04 on Lenovo ThinkPad x240 with a Core i5 @ 2.4GHz and 4GB of RAM. The Android version is 4.1.1_r6 and Linux kernel version is 2.6.29 with XATTR supported. We describe the results by the comparison between DDDroid and a corresponding AOSP Android to measure the memory and performance overheads introduced by DDDroid. Both the DDDroid and AOSP Android images include none additional third-party application in our evaluation.

A. Evaluation 1

Proportional set size (PSS) is used to measure the memory usage of each application, where shared memory pages are divided by the number of processes sharing them. Therefore, PSS is a good measure for RAM usage comparison between different applications. In AOSP Android, we recorded the value of PSS of each application after the boot was completed (no manual operation is done to the emulator). In DDDroid, We started applications DDNotify, DDPolicy and DDFile (three applications running in the background), and then pressed the home button after the boot was completed (no other manual operation is done to the emulator) in DDDroid. We then recorded the value of PSS of each application. Figure 11 shows the results of memory usage of AOSP Android and DDDroid. DDDroid has 8.5% lead in memory consumption. More specifically, application DDNotify totally costs 4591KB, application DDPolicy totally costs 14647KB, and application DDFile totally costs 5521KB. What'more, appliation DDPolicy and DDFile are not used frequently and they expended the 4.7% of memory consumption.

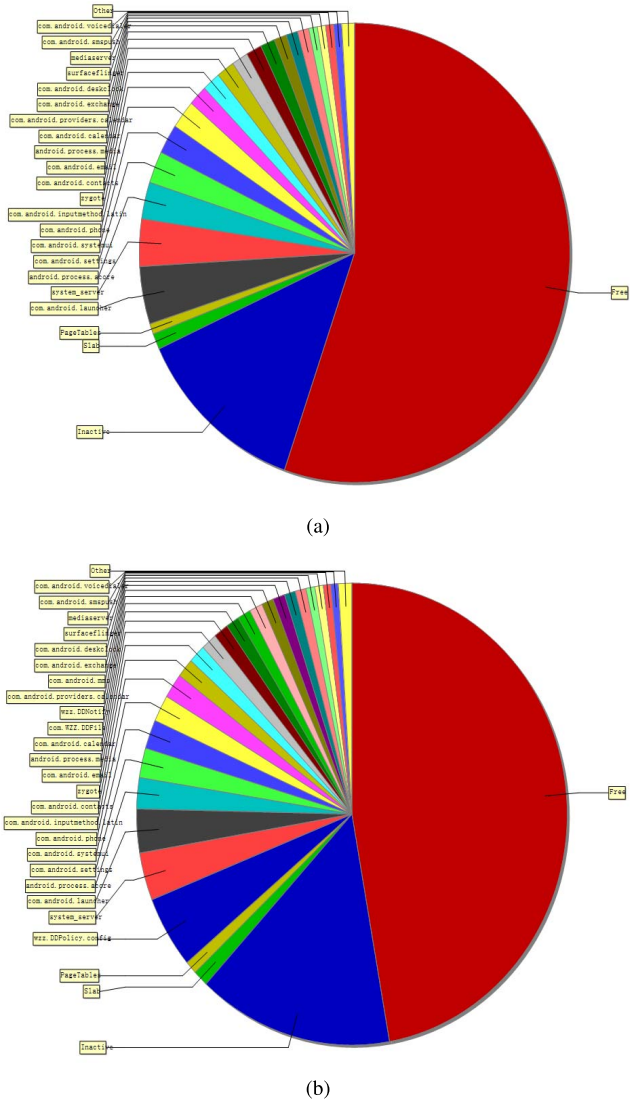


Fig. 11. The comparison of memory usage between AOSP Android and DDDroid. (a) The memory usage of each application in AOSP Android. (b) The memory usage of each application in DDDroid.

We use command "adb shell dumpsys meminfo" to measure the detail memory usage information of each application. We find that the increment of memory consumption is largely

TABLE II
AN COMPARISON OF AOSP ANDROID AND DDDROID ON HIGH-LEVEL PHONE OPERATIONS

	AOSP Android (ms)	DDDroid (ms)	Delay (%)
Call phone	276	302	9
Send SMS	675	830	23
Read contacts	432	496	15
Write contacts	753	901	20
Take picture	2453	2920	19
Read file	1237	1412	14
Write file	2676	3157	18

introduced by the category "Dalvik"and ".so mmap". This is because we stored 32 taint markings for each 32-bit variable in Dalvik, and we added libLabel.so for each application to set and get their labels at runtime.

B. Evaluation 2

We use two well-known benchmark applications: Benchmark [1] and CaffeineMark,[2] to measure the runtime overhead of DDDroid. First, we run each workload in Benchmark (version 1.0) on DDDroid and AOSP Android 20 times. The graphic performance (MPixels per sec) and memory performance tests averagely incur only 2.4% overhead. On the CPU tests, DDDroid have round 19% overhead due to the dynamic label marking, tracking, modulating and controlling at runtime. We then run each workload in CaffeineMark (version 3.0) on DDDroid and AOSP Android 20 times. CaffeineMark's score is roughly correspond to the number of Java instructions executed per second. The overhead incurred by arithmetic is the smallest (8%) and by string is the greatest (26%). This is because label tracking for arithmetic computing is simple and additional memory comparisons for string objects. The overall overhead incurred by DDDroid is 18% with respect to the AOSP Android system.

To gain further insight into perceived overhead on common high-level smartphone operations, we measured the costs of calling phone, sending SMS, reading and writing contacts (total five SQL transactions), taking picture (average size 420KB), reading and writing file (a. doc file with size 2MB). Table II shows that DDDroid averagely brings 17% overhead with respect to the Android system. This is because the additional time is needed to do following operations: get and set the label of applications, variables and files, and enforce the label related security policies.

VII. DISCUSSION AND FUTURE WORK

In this paper, we introduced the four key minds: dynamic label marking, dynamic label tracking, dynamic label modulating, and run-time monitoring and controlling. However, there are a lot of works to do to enrich these four key minds. For example, how to decide the initial value of the *risk* label more precisely. How to track the label between different boundaries such as OS, database, network and so on. We also implemented DDDroid which is based on Dalvik VM. Note that Dalvik is replaced by ART nowadays. In future, we will implement our model in ART environment by combining with the work that implemented taint tracking on ART [51], [52]. In our prototype, we only concern the explicit information flow caused by data dependence, while ignore the impact of implicit information flow (IIF) caused by control dependence. As a result, it is possible for an attacker to leverage implicit information flow to evade our detection. For example, considering the following codes:

if (*high* == 0) **then** *low* = 0; **else** *low* = 1;

Although there is no direct information flow between variable *high* and *low*, one can still learn some information about *high* by observing *low*. You *et al.* [53] called this kind of implicit information flow as *if*-based IIF. Other kinds of implicit information flow include *switch*-based IIF, *exception-prone*-based IIF, *throw*-based IIF and *polymorphism*-based IIF. Obviously, the amount of information leaked by implicit information flow is much smaller than explicit information flow. Therefore, quantitative information flow can be used to determine whether track this flow or not. TaintMan [53] proposed a method to track the strict control dependence flow and SpanDex [54] described a way to track implicit flow on password. These methods can be applied to our system in the future. We also want to establish a taint propagation framework to enable further taint tracking optimization on hot loop traces to make the system more effective. Thresholds defined in DDDroid are manually configured currently, adding adaptive algorithm for self-tuning of these thresholds could be a future work. At last, constructing more precise and efficient machine learning algorithms for dynamic behavior evaluation is also a future direction.

VIII. CONCLUSION

It is a great challenge when we want to use suspicious data or codes from not fully trusted Internet source as well as keep the operating system secure. To address this problem, we present a novel dynamic defense model to reduce security risks brought by the suspicious data or codes. DDM is a high-level security defense abstraction with the four key minds: dynamic label marking, dynamic label tracking, dynamic label modulating, and run-time controlling. With these minds, DDM provides an abstract framework to deal with the potential security risks through integrating the existing security mechanisms together organically. We also implemented DDDroid, a prototype system of DDM on Android that provides a robust dynamic security defense mechanism for smartphone users. We evaluated our prototype on functions and performance through several experiments. The results show that DDDroid is an accurate, flexible and available system without sacrificing some aspects of programmability, compatibility, convenience and performance.

ACKNOWLEDGMENT

The authors would like to thank Prof. Lorenzo Cavallaro and reviewers for their valuable comments and suggestions. They would like to thank Jian Li, an associate professor of Beijing University of Posts and Telecommunications, for his help of

[1] https://www.apkshub.com/app/softweg.hw.performance
[2] http://www.benchmarkhq.ru/cm30/

getting the Android malware datasets and providing sagacious comments. They would also like to thank Hao Chen for his help on the English language.

REFERENCES

[1] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang, "Design and implementation of an Android host-based intrusion prevention system," in *Proc. ACM 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New York, NY, USA, 2014, pp. 226–235.

[2] S. Smalley and R. Craig, "Security enhanced (SE) Android: Bringing flexible MAC to Android," in *Proc. NDSS*, vol. 310. 2013, pp. 20–38.

[3] P. Efstathopoulos *et al.*, "Labels and event processes in the asbestos operating system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 17–30, 2005.

[4] M. Krohn and E. Tromer, "Noninterference for a practical DIFC-based operating system," in *Proc. 30th IEEE Symp. Secur. Privacy*, Jul. 2009, pp. 61–76.

[5] C. Li, A. Raghunathan, and N. K. Jha, "Secure virtual machine execution under an untrusted management OS," in *Proc. IEEE Int. Conf. Cloud Comput.*, Jul. 2010, pp. 172–179.

[6] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Proc. Workshop Hot Topics Softw. Defined Netw.*, 2012, pp. 127–132.

[7] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, 1997.

[8] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS—1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Exper. Returns Secur.*, 2014, pp. 3–17.

[9] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, p. 76, 2017.

[10] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2010.

[11] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A user-oriented behavior-based malware variants detection system for Android," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1103–1112, May 2016.

[12] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2016.

[13] L. Jia *et al.*, "Run-time enforcement of information-flow properties on Android," in *Proc. Eur. Symp. Res. Comput. Secur.* Berlin, Germany: Springer, 2013, pp. 775–792.

[14] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC enforcement on Android," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 1119–1136.

[15] F. Martinelli, I. Matteucci, M. Petrocchi, and A. Saracino, "Risk analysis of Android applications: A multi-criteria and usable approach," Nat. Res. Council, Italy, Tech. Rep., 2015.

[16] Y. Wang, J. Zheng, C. Sun, and S. Mukkamala, "Quantitative security risk assessment of Android permissions and applications," in *Data and Applications Security and Privacy XXVII*. Berlin, Germany: Springer, 2013.

[17] R. Sekar *et al.*, "Model-carrying code: A practical approach for safe execution of untrusted applications," in *Proc. ACM SIGOPS Oper. Syst. Rev.*, 2003. pp. 15–28.

[18] H. V. Nath and B. M. Mehtre, "Static malware analysis using machine learning methods mehtre," in *Proc. Int. Conf. Secur. Comput. Netw. Distrib. Syst.*, 2014, pp. 440–450.

[19] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2008, pp. 1939–1945.

[20] X. Chen, J. Andersen, Z. M. Mao, J. Nazario, and M. Bailey, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, Jun. 2008, pp. 177–186.

[21] W. Sun, Z. Liang, V. N. Venkatakrishnan, and R. Sekar, "One-way isolation: An effective approach for realizing safe execution environments," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2005, pp. 265–278.

[22] M. Payer, T. Hartmann, and T. R. Gross, "Safe loading—A foundation for secure execution of untrusted programs," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 18–32.

[23] J. Huang, Z. Jiang, and R. Akhter, "Protection tiers and their applications for evaluating untrusted code on a Linux-based Web server," *J. Commun.*, vol. 10, no. 11, pp. 1891–1899, 2015.

[24] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical taint-based protection using demand emulation," *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 29–41, 2006.

[25] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry, "Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools," *ACM SIGPLAN Notices*, vol. 45, no. 6, pp. 25–35, 2010.

[26] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 235–246.

[27] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Apr. 2008, pp. 74–83.

[28] Z. Wei and D. Lie, "LazyTainter: Memory-efficient taint tracking in managed runtimes," in *Proc. ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2014, pp. 27–38.

[29] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 121–132, 2012.

[30] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2006, pp. 135–148.

[31] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *Proc. NDSS*, 2012, pp. 124–137.

[32] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. (2005). *JIF: JAVA Information Flow, Softw. Release*. [Online]. Available: http://www.cs.cornell.edu/jif

[33] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," *Commun. ACM*, vol. 54, no. 11, pp. 263–278, 2011.

[34] W. Cheng *et al.*, "Abstractions for usable information flow control in Aeolus," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2012, pp. 139–151.

[35] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *Proc. NSDI*, vol. 8, 2008, pp. 293–308.

[36] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 321–334.

[37] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proc. ACM Laminar, Practical Fine-Grained Decentralized Inf. Flow Control*, 2009, vol. 44, no. 6, pp. 63–74.

[38] M. Xu *et al.*, "Toward engineering a secure Android ecosystem: A survey of existing techniques," *ACM Comput. Surv.*, vol. 49, no. 2, p. 38, 2016.

[39] S. K. Dash *et al.*, "DroidScribe: Classifying Android malware based on runtime behavior," in *Proc. Secur. Privacy Workshops*, 2016, pp. 252–261.

[40] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of Android malware behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Aug. 2015, pp. 241–245.

[41] B. Rashidi, C. Fung, A. Ngyun, T. Vu, and E. Bertino, "Android user privacy preserving through crowdsourcing," *Trans. Inf. Forensics Secur.*, vol. 13, no. 3, pp. 773–787, Mar. 2017.

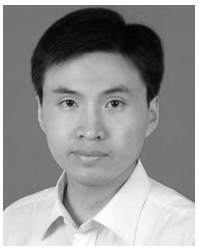
[42] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate Android malware," in *Proc. 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jul. 2013, pp. 163–171.

[43] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated Android malware," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 309–320.

[44] G. Smith, "On the foundations of quantitative information flow," in *Proc. Int. Conf. Found. Softw. Sci. Comput. Struct. Held As*, 2009, pp. 288–302.
[45] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 482–493, Jun. 2007.
[46] D. Schultz and B. Liskov, "Ifdb: Decentralized information flow control for databases," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, Aug. 2013, pp. 43–56.
[47] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce," in *Proc. Usenix Symp. Networked Syst. Des. Implement.* San Jose, Ca, USA, Aug. 2010, pp. 297–312.
[48] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012.
[49] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.
[50] D. Arp, M. Spreitzenbarth, M. Hábner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Aug. 2014, pp. 23–26.
[51] M. Sun, T. Wei, and J. C. S. Lui, "Taintart: A practical multi-level information-flow tracking system for Android runtime," in *Proc. ACM Sigsac Conf.*, Oct. 2016, pp. 331–342.
[52] M. Backes, O. Schranz, and P. von Styp-Rekowsky, "POSTER: Towards compiler-assisted taint tracking on the Android runtime (ART)," in *Proc. ACM Sigsac Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 1629–1631.
[53] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices," *IEEE Trans. Dependable Secure Comput.*, to be published.
[54] L. P. Cox *et al.*, "Spandex: Secure password tracking for Android," in *Proc. USENIX Conf. Secur. Symp.*, Aug. 2014, pp. 481–494.

Zezhi Wu received the Master degree in communication and information system from the Institute of Zhengzhou Information Science and Technology, Zhengzhou, China, in 2015, where he is currently pursuing the Ph.D. degree. His research interests include information flow control and android security.

Xingyuan Chen received the Ph.D. degree in communication and information system from the Institute of Zhengzhou Information Science and Technology, Zhengzhou, China, in 2003. He is currently a Professor and a Doctoral Supervisor with the Institute of Zhengzhou Information Science and Technology. He is also a Doctoral Supervisor with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. His research interests are generally in the areas of cyberspace security, cloud computing security, and OS security.



Zhi Yang received the Ph.D. degree in computer science and technology from the Chinese Academy of Sciences, Beijing, China, in 2012. He is currently an Associate Professor with the Institute of Zhengzhou Information Science and Technology, Zhengzhou, China. His research interests include information flow control and OS security.

Xuehui Du received the Ph.D. degree in computer science and technology from the Institute of Zhengzhou Information Science and Technology Institute, Zhengzhou, China, in 2011. She is currently a Professor and a Doctoral Supervisor with the Institute of Zhengzhou Information Science and Technology. Her research interests include cyberspace security and android security.