D2Taint: Differentiated and Dynamic Information Flow Tracking on Smartphones for Numerous Data Sources

Boxuan Gu, Xinfeng Li, Gang Li, Adam C. Champion, Zhezhe Chen, Feng Qin and Dong Xuan
Dept. of Computer Science and Engineering
The Ohio State University
{gub, lixinf, lgang, champion, chenzhe, qin, xuan}@cse.ohio-state.edu

***Abstract*—With smartphones' meteoric growth in recent years, leaking sensitive information from them has become an increasingly critical issue. Such sensitive information can originate from smartphones themselves (e.g., location information) or from many Internet sources (e.g., bank accounts, emails). While prior work has demonstrated information flow tracking's (IFT's) effectiveness at detecting information leakage from smartphones, it can only handle a limited number of sensitive information sources. This paper presents a novel IFT tagging strategy using differentiated and dynamic tagging. We partition information sources into differentiated classes and store them in fixed-length tags. We adjust tag structure based on time-varying received information sources. Our tagging strategy enables us to track at runtime numerous information sources in multiple classes and rapidly detect information leakage from any of these sources. We design and implement D2Taint, an IFT system using our tagging strategy on real-world smartphones. We experimentally evaluate D2Taint's effectiveness with 84 real-world applications downloaded from Google Play. D2Taint reports that over 80% of them leak data to third-party destinations; 14% leak highly sensitive data. Our experimental evaluation using a standard benchmark tool illustrates D2Taint's effectiveness at handling many information sources on smartphones with moderate runtime and space overhead.**

I. Introduction

A. Motivation

Smartphones are becoming increasingly popular. According to Nielsen data from July 2012, 54.9% of U.S. mobile users own smartphones and two out of three new handset buyers bought a smartphone in the past three months [1]. This is mainly due to smartphones' all-in-one features combining communication and computing functions, enabling a wide variety of applications (also referred to as *apps*).

As smartphones become more widespread, their users' privacy and security become critical issues. For example, a *Wall Street Journal* study of iOS and Android applications revealed that 46–55% of smartphone applications transmit users' private information such as location and device ID over networks without users' awareness or consent [2]. Worse, many users are enticed to download and run smartphone applications without carefully understanding the consequences of accepting permissions prompted before installation. This can easily lead to installation of malicious applications. In fact, Trend Micro reports that over 25,000 Android malware samples were found in June 2012 alone [3].

This research was partially supported by China 973 Project 2011CB302800 and the U.S. National Science Foundation (NSF) under grants No. CCF-0953759 (CAREER), CCF-1218358, CNS-0916584, CNS-1065136 and CNS-1218876. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

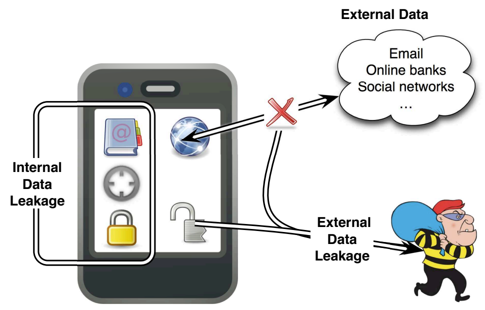


Fig. 1: Information leakage in smartphones

Private (or sensitive) information on smartphones comes from various sources, including sources originating from smartphones themselves and sources received from the Internet. Fig. 1 illustrates this sensitive information and its leakage. On one hand, smartphones themselves generate sensitive information such as photos, GPS locations, and device identifiers (IMEIs/EIDs). On the other hand, smartphones can receive sensitive information from a plethora of possible sources over the Internet. For example, users may check their bank accounts via a browser or a bank-provided application. Similarly, smartphones are often used for checking email contents from servers such as Gmail or Yahoo! Mail. Privacy can be easily invaded if sensitive data from one source were sent to another irrelevant destination, let alone an attacker-controlled one.

Prior work such as TaintDroid [4] has demonstrated that Information Flow Tracking (IFT) mechanisms can be leveraged to detect leakage of sensitive information. Specifically, TaintDroid extended the Dalvik virtual machine (VM) to tag smartphone data using 32 possible types based on their origins, to propagate tags during program execution, and to alert users when leakage of sensitive data is detected. While effective at tracking a limited number of sensitive data sources, TaintDroid's scheme cannot scale to handle sensitive data received from many possible external sources such as Citibank, Bank of America, and Gmail. For example, after "normal" use in which a malicious browser accesses a bank website, the browser may send the bank information to a predefined attacker's server. In this scenario, TaintDroid tags the account information data as "network", indicating it originates from the network, and may notify the user before the data are sent to another site. However, with the general tag "network", rather than the data's precise source, it is difficult for users to determine whether sending the data is legitimate.

Therefore, it is imperative to design an IFT system for tracking sensitive data from a large number of possible internal and external sources on smartphones and informing users via alerts with relevant source and destination information before data are sent out.

However, our key challenge is tracking a vast number of information sources given limited resources. Smartphone data can originate from many sources such as online banks, social networking websites, etc. Any smartphone based IFT system needs to track all these sources in each data tag. Yet tag capacity is limited, e.g., 32 bits. A naïve approach to solve this challenge is using one bit to track each source. But this requires many bits to track all sources, far more bits than the tag length. This approach leads to enormous tag overhead. Also, tracking data during program execution is not lightweight as every statement requires tag propagation. The runtime overhead may be even higher if a compressed tag system (for saving space overhead) is used. Crucially, our tracking system cannot be so slow as to exceed users' expectations on response times.

B. Contributions

This paper proposes D2Taint, a novel IFT tagging strategy using differentiated and dynamic tracking. We partition information sources into disjoint *classes* that correspond to different information sensitivities. We design a flexible tag structure that stores these classes and their information sources in fixed-length tags. Our tag structure updates itself on-the-fly based on time-varying received information sources. Our tagging strategy enables us to track at runtime numerous information sources in multiple classes and rapidly detect information leakage from any of these sources.

In summary, our contributions are as follows:

– We propose a novel IFT strategy using differentiated and dynamic tagging. With its flexibility, our tag scheme can handle different numbers of information sources, from a few to thousands;

– We leverage our tagging strategy to design and implement D2Taint, an IFT system using differentiated and dynamic tagging on Nexus One smartphones running Android 2.2;

– We experimentally evaluate D2Taint's effectiveness with 84 real-world applications downloaded from Google Play [5]. D2Taint reports that 71 out of the 84 evaluated applications leak either internal or external information to third-party destinations and 12 out of these 71 applications leak highly sensitive internal information. D2Taint also detects considerable external information leakage in 33 out of these 71 applications and provides detailed information about multiple external sources, which is much more than TaintDroid can provide. Furthermore, we evaluate the performance of D2Taint and our dynamic tag system. Our results show that D2Taint is effective at handling a large number of sources and it can dynamically adjust its tag scheme based on users' behavior during program execution with moderate space and runtime overheads. For instance, using the CaffeineMark benchmark, D2Taint's space overhead is 4.0%, which is slightly less than TaintDroid's. D2Taint's runtime overhead is 25.9% while TaintDroid's is 14%. This is expected, as D2Taint needs more operations for the tag system and location information table operations.

The rest of the paper is organized as follows. Section II provides background information on the Android system and IFT. Section III presents our differentiated and dynamic tagging strategy. Section IV presents the design and implementation of D2Taint, respectively. Section V describes our evaluation methodology. Section VI discusses our experimental results. Section VII provides related work. Section VIII concludes.

II. Background

A. Android System

Android [6] is an open, Linux-based mobile operating system for which applications are written in Java. They are compiled to Java bytecode, which is translated to custom Dalvik EXecutable (DEX) bytecode for the Dalvik virtual machine (VM). Dalvik is a register-based VM that interprets Android applications' DEX bytecode. Applications communicate with each other using Binder IPC, which enables message passing via parcels.

B. Information Flow Tracking Basics

Information flow tracking (IFT) is a promising and effective technique for detecting leakage of sensitive data [7]–[9] and system-compromising security attacks [10]–[12]. It can be implemented in three different ways, including compiler analysis on programs written in special type-safe programming languages [7], [8], [13]–[15], software instrumentation at the source code, bytecode, or binary level [4], [11], [16], and architectural support for IFT [9], [12], [17].

To detect leakage of sensitive data, IFT techniques generally tag (label) the source data using a pre-defined structure, e.g., sensitivity level or source IDs. The source data can come from I/O devices such as disks, keyboards, and cameras. For example, data in the password file can be tagged with the highest sensitivity or the file's owner ID (`root`). During program execution, data tags are propagated based on certain policies. For example, $\texttt{a} = \texttt{b} + \texttt{c}$ means `a`'s tag derives `b` and `c`'s tags. One propagation policy could be $\texttt{a.tag} = \max(\texttt{b.tag}, \texttt{c.tag})$ if we use the sensitivity level as the tag. Finally, the data tag is checked against security rules whenever certain data are sent over channels like networks. If a security rule is violated, an information leakage alarm is raised. An example of such a rule could be "The most sensitive data (e.g., passwords) may not be sent over the network."

III. Differentiated and Dynamic Tagging

A. Design Rationale

Recall from Section I-B that our key challenge is tracking a vast number of information sources. Smartphone data can originate from many sources such as online banks, social networking websites, etc. Any smartphone based IFT system needs to track all these sources in each data tag. Yet tag capacity is limited, e.g., 32 bits. A naïve approach to solve this challenge is using 1 bit to track each source. However, this requires many bits to track all sources, far more bits than the tag length. This approach leads to enormous tag overhead. On the other hand, we aim to achieve two contradictory goals: *source completeness* and *accuracy of source information*. Source completeness refers to how many information sources we capture. Clearly, we want to capture as many sources as possible. Accuracy of source information refers to the accuracy with which we map information recorded in the tag to the specific information source that was recorded. This information recorded in the tag should *uniquely* identify this information source. But this may be infeasible due to the limited tag length. A better solution must be sought.

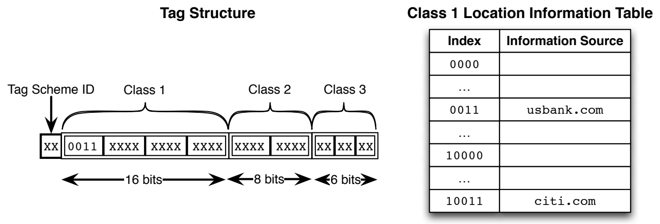
In this paper, we propose a so-called differentiated and dynamic tagging strategy to overcome the above challenge while balancing these two contradictory goals. Our strategy is based on the following observations at three different levels: information sources, applications, and user behaviors.

Source Level: Different information sources may have different sensitivities in terms of security. For example, information from online bank websites is far more sensitive than information from news websites. As such, bank information merits a higher level of security than news information.

Application Level: The patterns by which applications access information sources vary for different applications. That is, these patterns follow a *non-uniform* distribution over the set of all applications. Some applications such as online banking ones access only a few sources whereas others such as news aggregators access many sources. Additionally, applications have variables that have different correlations with each other. For instance, if we execute the statements $\texttt{a} := \texttt{c}+\texttt{d}+\cdots+\texttt{z}$ and $\texttt{b} := \texttt{1}$, a correlates with variables $\texttt{c}, \ldots, \texttt{z}$ whereas b correlates with no other variables. Clearly, we need to capture these heterogeneous variable correlations. Thus, we need different amounts of storage space (bits) to capture heterogeneous sources and correlations.

User Level: We observe that smartphone users' behavior patterns vary over time. Consider the following real-world usage scenario. Suppose a user often reads many news websites like `cnn.com`. Once or twice, the user logs in a social networking website like Facebook to check for messages from friends and checks a bank account. From this scenario, we can see that users access different information sources over time. Thus, IFT needs to adapt to changing information source access patterns.

We develop the differentiated and dynamic strategy based on these levels. By examining the source level and application level, we develop *differentiated classes*, which classify information sources based on considerations such as their information sensitivities. Differentiated classes inform us about information sources' sensitivity. This information provides guidance for designing the tag scheme given limited tag length. By examining



Index	Information Source
0000	
...	
0011	usbank.com
...	
10000	
...	
10011	citi.com

Fig. 2: Tag structure

the user level, we develop tag *dynamics*, in which the tag scheme is updated on-the-fly based on properties of received information sources and users' access thereof. Differentiated classes and tag dynamics enable us to track many information sources on smartphones in real-time. We discuss differentiated classes in Section III-B and tag dynamics in Section III-C.

B. Differentiated Tag Structure

Our tag structure is illustrated in Fig. 2. Each tag has the same fixed length. There may be multiple tag schemes in our tag system when dynamics are considered (we discuss dynamics shortly). We assign each tag a tag scheme ID, which is a fixed length bit string at the beginning of the tag. We partition the remainder of the tag into segments. Each segment corresponds to a distinct class. Each segment contains a certain number of information sources. Intuitively, there are fundamental tradeoffs among the number of classes in a tag, the number of information sources that can be stored in a class, and the number of bits used to represent an information source in that class. In fact, the bits representing a source map to the indices of a location information table. Each class has its own table. For a particular class, each entry in that class's table "describes" an information source in that class. Specifically, each table entry maps the bits representing an information source to a text string describing that source. As a concrete example, consider the tag in Fig. 2. We see the tag is 32 bits long with a 2-bit tag scheme ID. We notice the remainder of the tag is partitioned into three segments of lengths 16 bits, 8 bits, and 6 bits. Each segment corresponds to a distinct class. In the first class, each information source is represented as a 4-bit string. In particular, the bit string `0011` maps to `usbank.com` in the location information table. Yet as more information sources arrive, the table grows and this string can map to either `usbank.com` or `citi.com`. In the second class, each information source is represented as a 4-bit string, and in the third class, each information source is represented as a 2-bit string. (These classes' tables are omitted for brevity.) Notice that source representations can have various numbers of bits in different classes.

Examples: The following examples illustrate our differentiated tag structure:

– *32 bits, 1 class for each bit:* This example assumes there is only a single tag, so there is no tag scheme ID field. In this example, each bit in a 32-bit tag can represent one information source. This is exactly the approach TaintDroid [4] uses. A tag system using this approach can support 32 distinct sources and keep 32 live records in the tag.

– *32 bits, 2-bit tag scheme ID, 3 classes, 16/8/6 bits per class, 4/4/2 bits per source:* This example shows the tag structure in Fig. 2. We have three classes: “highly sensitive”, “moderately sensitive”, and “insensitive”. We allocate 16 bits for “highly sensitive”, 8 bits for “moderately sensitive”, and 6 bits for “insensitive”.

– *32 bits, 2-bit tag scheme ID, 2 classes, 24/6 bits per class, 3/2 bits per source:* In this example, we allocate 24 bits for the “highly sensitive” class and 6 bits for the “insensitive” class. We can store 8 sources in the “highly sensitive” class and three sources in the “insensitive” class. This example works for the case where there are more sensitive sources.

Tag Parameter Settings: We can realize differentiated classes based on information sensitivities. We can change the number of classes so long as each class stores at least one information source representation and the tag length is fixed. Clearly, we face tradeoffs among the class length, number of information sources represented in each class, and the number of bits for each source representation in the tag. Our proposed design incurs the following tradeoffs among the number of classes, number of information sources, and bits per source.

– If we record too few sources in the tag, some sources will be absent from the tag. If we use too few bits to represent a source, collisions can occur among sources. Then there will be less accurate source information within a class.

– There are two special cases within a class: (1) Many sources with very few bits per source. This works well when an application accesses few information sources but its individual variables are correlated with many other variables; (2) Very few sources with many bits per source. This works well when an application accesses many different information sources but its individual variables are only correlated with a few variables.

– We cannot arbitrarily set the number of classes, number of sources, and bits per source. The total number of bits is bounded by the tag space. Also, the number of sources in a class is at most 2^n, where n is the number of bits per source.

C. Tag Dynamics

We realize tag dynamics as follows. Each class can have a different length at different times. As new information sources arrive, we classify them based on sensitivity, add them to the respective location information table, and place their (truncated) indices in the tag. Based on information source knowledge, we can adjust the class size for each class. An intuitive way is to “pre-specify” some classes and change the tag structure once certain conditions are met, e.g., most tags have less than 50% space usage. Another way is to perform “on-demand” machine learning (ML) based on statistical properties of tag space usage and location information tables’ recent hash values. With this way, we create an ML process, which collects tag information from a normal process, and calculates a new tag structure for the normal process. After a new tag structure is specified, the ML process sends the new tag structure to the normal process, which can adjust its tag structure accordingly.

In designing tag dynamics, we need to consider how the tag structures should be changed. Initially, the tag should record as many sources per class as possible subject to the constraints described above. If most variables do not use their entire class space during tag scheme system execution, we can assign more bits per source and adjust class length accordingly. For such an adjustment, we need to consider the following two issues:

– *Tag Scheme Switching:* Switching among different tag schemes is crucial for our dynamic tag system design. There are two problems: (1) determining tag scheme configurations; and (2) determining when to switch tag schemes. We have two main approaches: (1) *pre-configured*; and (2) *on-the-fly*. The pre-configured approach lets users configure their own tag schemes based on their Internet usage behaviors. Recall the last two examples in Section III-B. The tag structure can switch between these when users access many highly sensitive information sources. For example, they can list important websites they often visit and classify them into different classes based on information sensitivities. All other websites and external sources are classified as “unknown.” When users access more sensitive websites than unknown sites, the source lengths in the sensitive class can be increased and the source lengths in the unknown class can be shortened. Criteria for switching and the switching process can be configured in advance. The on-the-fly approach can adjust the tag scheme based on tag space utilization and newly arriving information sources. It is suitable for determining when to switch tag schemes as user and application behaviors are unpredictable. For example, if class 2 sources arrive while class 1’s space is not full, the system dynamically allocates more space for class 2.

– *Tag merging:* Tag merging is necessary for tag propagation. When multiple tags “meet” based on variable operations, we need to generate a new tag. If both merging tags have different tag schemes due to tag switching, we first need to convert old tags to new tags. In a simple case, both tag schemes’ representations have the same length and we can simply merge them in the new tag scheme. If we cannot fit all information sources in the new class, we can select source representations for replacement. But their lengths may differ. If the length decreases in the new tag system, some most significant bits need to be removed from the old representations. If the length increases, we retrieve “old” indices from the table and place them into the new class segments.

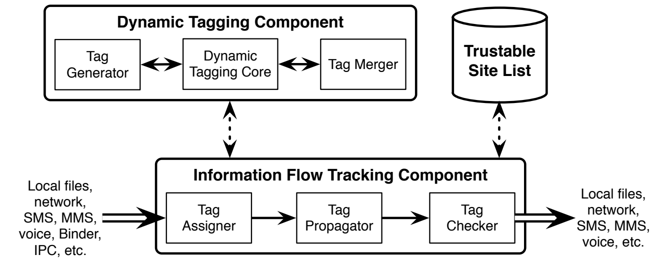

Dynamic Tagging Component
Tag Generator
Dynamic Tagging Core
Tag Merger
Trustable Site List
Information Flow Tracking Component
Local files, network, SMS, MMS, voice, Binder, IPC, etc.
Tag Assigner
Tag Propagator
Tag Checker
Local files, network, SMS, MMS, voice, etc.

Fig. 3: D2Taint system architecture

IV. IFT WITH DYNAMIC AND DIFFERENTIATED TAGGING

A. System Overview

Fig. 3 shows our D2Taint system architecture. The system has two main components: (1) the *dynamic tagging* component; and (2) the *information flow tracking* component.

The dynamic tagging component handles tag management and determines when to switch tag schemes. It has three parts:

(1) the *dynamic tagging core*; (2) the *tag generator*; and (3) the *tag merger*. The dynamic tagging core handles configuration of the dynamic tag system and decides which tag scheme can be used. The tag generator fetches or generates a tag for incoming data. The tag merger merges two tags and generates a new tag.

The information flow tracking component tracks a data flow from its sources until the related data are sent out or written to files. It has three parts: (1) the *tag assigner*; (2) the *tag propagator*; and (3) the *tag checker*. The tag assigner intercepts incoming data via I/O channels such as networks and assigns the initial tag to the data based on the results from the dynamic tagging component. The tag propagator propagates data tags for each operation during program execution. The tag checker checks the data tags for compliance with security policies when data are going to be sent over I/O channels. Our system also maintains a trustable site list containing a list of websites to which data can be sent without raising any alerts. Users can modify the list based on the IFT results.

A typical D2Taint workflow is as follows. When an application starts, the dynamic tagging component first loads two configuration files: one stores tag structure definitions; the other stores user-defined classes and known data sources in each class. Then the dynamic tagging component performs two tasks. First, this component checks the data source list for each incoming data source passed by the tag assigner. If found, the tag is retrieved and returned to the tag assigner. Otherwise, a new entry for the data source is created and the new tag is returned. Second, the dynamic tagging component tracks incoming sources' statistics and determines whether it should switch D2Taint to a different tag scheme.

The tag assigner intercepts I/O channels to capture incoming data. For each incoming data source, the tag assigner consults the dynamic tagging component to get the tag and then assigns the tag to the new data.

For each instruction being executed, the tag propagator is in charge of propagating tags for a data flow. If it needs to merge multiple tags to generate a new tag, it gets a new tag from the dynamic tagging component. The dynamic tagging component merges the tags from different source data and assigns the merged tag to the destination data. For example, a binary operation $\texttt{a} = \texttt{b} + \texttt{c}$ triggers tag propagation $\texttt{a.tag} = \texttt{b.tag} \oplus \texttt{c.tag}$, where $\oplus$ is the merge operation. The dynamic tagging component handles tag merging differently for two cases: (1) where the tag scheme is not switched; and (2) where `b.tag` and `c.tag` use different tag schemes.

When data are going to be sent over I/O channels, the tag checker is activated. It uses the list of trustable sites to check if the data are allowed to be sent to their destinations. If this data sending event is not allowed, the tag checker raises an exception to the user, who is asked if the data can be sent.

We implement a prototype of D2Taint on Nexus One smartphones running Android 2.2. However, we see no particular difficulty applying our ideas to other smartphone architectures. To reduce overhead incurred by tag storage and propagation, our D2Taint system tracks data flows for Java code at the variable level, as TaintDroid [4] does. By instrumenting the Dalvik VM during interpretation of Java bytecode, we can fully utilize Java objects' semantics to store taint tags and propagate them among objects. This helps reduce IFT's overhead.

In the following, we detail these two components. First, we introduce the dynamic tagging component, then we describe the information flow tracking component.

B. Dynamic Tagging Component

The dynamic tagging component realizes management of tag schemes, tag assignment, and tag merging. These functions are performed by three sub-components: the dynamic tagging core, the tag assigner, and the tag merger, respectively. We discuss them in the following.

1) Dynamic Tagging Core: The dynamic tagging core maintains two configuration files. The first configuration file stores the tag system settings, which are read and parsed into memory during D2Taint's initialization phase. In memory, we maintain a global array that stores settings for each tag scheme. Each tag scheme setting includes the scheme number, the number of bits per tag, the number of classes, and a pointer to the class list. In a class structure, we record the number of classes in the tag system, the number of bits per hashcode, the number of reserved slots for the class, and a text description of the class. In our current implementation, we use 32 bits for a tag and 5 bits for a hashcode by default. The first 2 bits are used to indicate the number of the tag scheme. There are 6 available hashcode slots in a tag.

The second configuration file stores user-defined classes and each class's known data sources. After reading the data from the configuration file, we use a global location information table list to record all source information. Each information table corresponds to one class. Each source has an entry in one particular table. Note that we only store the domain name for a source, e.g., `google.com`, `nsf.gov`, etc. If an IP address has no corresponding domain name or hostname, we store the first 16 bits of the address, e.g., `192.168.0.0`. This helps decrease the total number of entries in the location information tables. A domain name or an IP address suffices for a user to determine the information source. To save space, each table is dynamically allocated as its number of entries increase.

The dynamic tagging core also collects statistics for incoming sources and determines whether the tag scheme should be switched. In particular, after a certain number of new sources (i.e., 50) are added into an location information table, D2Taint decides whether to switch the tag scheme based on these new sources. D2Taint counts the source distributions for each class, finds the best matched scheme with this distribution, and updates the current tag scheme number. This implementation incurs a low overhead as it only makes the "switch" decision periodically after enough new sources arrive.

2) Tag Generator: The tag generator uses the location information list to respond to the tag assigner's query when new data arrive. The tag generator checks the location information list for each new data source. If the source is in the list, the tag is retrieved and returned to the tag assigner. Otherwise, a new entry for the data source is created and a new tag is returned.

3) Tag Merger: The tag merger performs tag merging, which is necessary for tag propagation. When multiple tags "meet" in a corresponding bytecode instruction, a new tag has to be generated based on the source data tags. For example, `a = b+c` triggers `a.tag` = `b.tag` $\oplus$ `c.tag`. To merge tags `b.tag` and `c.tag`, we need to handle two cases: (1) when these two tags use the same tag scheme; and (2) when they use different tag schemes. Merging multiple tags can be seen as multiple instances of merging two tags.

For the first case, a new tag can be formed by collecting the hashcodes in the corresponding class segments when each class has enough room to host all hashcodes from `b.tag` and `c.tag`. Sometimes, the classes in `b.tag` and `c.tag` contain more sources that one class segment in `a.tag` can hold. To handle this case, we can either randomly drop sources or select source tags based on their access recency or frequency.

For the second case, we have to convert an old tag to a new tag based on the current tag scheme. In a simple case, the old tag scheme has the same hashcode length as the current tag scheme. If so, we simply place the hashcode into its class segment in the new tag. When there are more hashcodes than available slots for a class, we can either randomly select some hashcodes or keep the latest ones (i.e., those with larger values). But the hashcode lengths may differ among different tag schemes. If the length decreases in the current tag scheme, certain significant bits need to be truncated from the old hashcodes. If the length increases, we first retrieve the hashcode indices in the location information table, hash these indices into new hashcodes, and finally fit the hashcodes into the class segments.

C. Information Flow Tracking Component

The information flow tracking component tracks information flows from sources to destinations in an Android application at runtime. It includes three sub-components: (1) the *tag assigner*; (2) the *tag propagator*; and (3) the *tag checker*, which perform tag assignment, tag propagation, and tag checking, respectively. We first discuss how D2Taint stores tags for different data, then we discuss each sub-component. In the following, we call the memory block that is used to store tags a *taint map*.

1) Taint Map: In the Dalvik VM, five types of variables need taint maps to store their tags: method local variables, method arguments, class instance fields, class static fields, and arrays. Among these data types, method local variables and method arguments are stored in methods' stack frames. We store tags of class static fields and arrays into their representative objects. TaintDroid [4] does likewise. However, for the other three variable types, our taint maps differ from TaintDroid's. We do not store variables' tags adjacent to them in memory. Instead, we use specific taint maps for these variable types, as our system's tag lengths tend to change. Further details follow:

Method local variables and method arguments. We use a stack taint map to store tags for a method's local variables and arguments. A stack taint map differs from a method's stack frame. When the Dalvik VM allocates a stack frame for a method, our system allocates a stack taint map for it. The last element of the stack taint map is for the method's return value.

Class instance fields. Tags for class instance fields are stored in objects' taint maps. An object's taint map is stored in the memory area immediately after that allocated for the object.

2) Tag Assigner: The tag assigner labels data tags according to their origins. While the data are read, the tag assigner tries to determine the data's origin and uses such information to query the dynamic tagging component. After it receives the tag, the tag assigner labels the data with the tag. If the origin information contains multiple sources, then the tag can be used to locate multiple sources. To taint data effectively, we insert our tag assigner logic into file I/O, network I/O, sensor, and other library functions that read private information, e.g., device identifiers, call histories, etc. TaintDroid [4] also does so.

3) Tag Propagator: As an IFT system, D2Taint needs to instrument program execution to track data flows. Based on this, we use the same propagation logic as TaintDroid to propagate tags in interpreted code and native code [4]. Our system also propagates tags from one process to another via Binder IPC, and writes the data's source information into the file system if the data are written to local files. The biggest difference is TaintDroid's use of bitwise OR to merge two or more tags, whereas we use the method in Section IV-B3 to merge two tags. Also, when a message is sent via Binder IPC, our system extracts source information from the related tags and sends it with the message via IPC. After the receiver gets the message from IPC, it extracts the message's source information and uses it to get a tag for tainting the received data.

4) Tag Checker: The tag checker is activated when data are going to be sent via networks. First, the tag checker leverages the list of trustable sites to determine the destination's trustworthiness. If the destination is trustable, the data can be sent without raising any alerts. If the destination is not in the trustable list, our system extracts source information from the data's tag and then delivers it to the user, who decides if the data can be sent to the destination. If the user does not want the data to be sent to the destination, the tag checker blocks data sending and stops program execution; otherwise, the data are sent and execution continues.

V. Evaluation Methodology

We evaluate our D2Taint system in three ways: (1) a real-world application study; (2) system performance evaluation; and (3) dynamic tagging system evaluation.

In the real-world application study, we select 84 "top free" applications from Google Play [5] in July 2012. We believe these applications represent a cross-section of those in widespread use on Android smartphones. Since many applications are ad-supported, we believe they may potentially leak sensitive information to third parties, as prior work suggests [4], [18]. We download these applications, install them on Nexus One smartphones running our D2Taint system, and exercise application functionalities. We monitor application installation and execution to check if these applications leak information.

We collect system logs, IPC messages, and network traces from the phones using `adb logcat`. We verify the results using `tcpdump` on the Nexus One's WiFi interface. No Nexus One had a SIM card and Bluetooth was disabled; all network traffic went through WiFi. When applications read data from the Internet, we record the data's sources via tags as well as the destinations to which the data are sent. We inspect the relevant hostnames and corresponding IP addresses to remove false positives, e.g., two different IP addresses belonging to the same organization. For comparison purposes, we perform the same experiments on smartphones running TaintDroid 2.3.

In the system performance evaluation, we demonstrate that D2Taint's overhead is reasonable. We use an unmodified Android ROM as the basis of performance comparison. First, we test D2Taint's impact on user experience, especially execution time. We test other common smartphone operations, including system and networking ones. Second, we use a standard benchmark tool, CaffeineMark [19], to measure D2Taint's overhead. CaffeineMark reports scores of various features based on Java execution time. Memory overhead is measured via CaffeineMark's increased memory footprint on the D2Taint system.

Lastly, we evaluate the performance of our dynamic tag system design and show the benefit of such a design. We develop a test application to emulate two different usage patterns: sequential and random. We measure the number of sources recorded in the tags when the data are sent via network sockets, which shows a tag's effective space utilization. We demonstrate the performance improvement of our dynamic tag design in comparison to a static tag system.

VI. Experimental Results

In this section, we present the experimental results following the above evaluation methodology.

A. Real-world Application Study

D2Taint finds that 71 out of the 84 applications leak information to third-party destinations. D2Taint reveals the paths by which the information is leaked, whereas TaintDroid only reveals the final leakage destinations. In our experiments, we found 33 applications that transmit data among many various external sources, especially cloud computing services (e.g., Amazon Web Services). To reduce false positives, D2Taint uses the following rule: information flows whose sources and destinations are the same are treated as legal. In addition, D2Taint provides detailed information about multiple sources when reporting to the user. By contrast, TaintDroid cannot record any source information for external data since it only uses 1 bit to tag the data. There are two consequences: (1) TaintDroid triggers false positives whenever data flows from an external source to that same source, which we observed frequently during experiments; and (2) TaintDroid cannot keep track of data from multiple sources at once. Our experiments validate the real-world problem of external data leakage and show that D2Taint can be used to detect information leakage related to many external sources.

In addition, D2Taint detects that some applications send highly sensitive internal data such as IMEIs/EIDs to third parties, particularly ad and market research companies (e.g., `admob.com` and `flurry.com`). More specifically, D2Taint finds 12 applications leaking devices' IMEIs/EIDs: The Weather Channel, ESPN ScoreCenter, NavFree GPS, SWAT Army, Bible, Fruit Ninja Free, Coin Dozer, Yellow Pages, Scramble with Friends, Words with Friends, Funny Facts Free, and IQ Test. From this aspect, TaintDroid also reports these applications leaking sensitive information.

TABLE I: Macrobenchmarks

	Stock Android	D2Taint
App Load	53.19 ms	58.12 ms
Download (32.3 MB)	35.73 s	38.34 s
Web Load (`google.com`)	735 ms	856 ms
Web Load (`nytimes.com`)	1081 ms	1116 ms
HttpGet	658 ms	746 ms
Write File	7.96 ms	8.02 ms
Read File	1.21 ms	1.51 ms
Socket Send	5.24 ms	6.37 ms

B. System Performance Evaluation

1) Macrobenchmarks: Macrobenchmark results are shown in Table I. Each value is averaged over 30 runs.

Application load time: We measure the time needed to load a new Android application and display the UI. D2Taint's overhead with respect to stock Android is 9%.

Download time: We measure the time needed to download a 32.3 MB file from `google.com`. D2Taint's overhead is 7.3%.

Webpage load time: We measure webpage load time using a toy "Web view" application. Specifically, the time between a UI button press and the webpage completely loading is measured. Two types of webpages are tested: light text (`google.com`) and heavy text (`nytimes.com`). Table I shows D2Taint's overheads are 16% for `google.com` and 3% for `nytimes.com`. This can be explained as follows. `google.com` automatically redirects to a "mobile-friendly" webpage, leading to more webpage data caching operations, as recorded in D2Taint logs. Since D2Taint also outputs tags into the file when writing data, `google.com`'s overhead is larger than `nytimes.com`'s.

Input and output: Besides basic system and networking operations, we develop an application that reads data from the "top 100" websites hosted in the U.S. [20], writes the data to a file, reads 1,000 bytes from this file, and transmits the 1,000 bytes to a remote machine via a socket connection. Table I shows the results. Each value is averaged over the top 100 websites' data with 10 runs.

The networking input and output overheads are 13% (HttpGet) and 21% (socket transmission), respectively. The input overhead stems from the location information table query to assign a new tag to the input data. For the output overhead, D2Taint needs to access the location information tables to look up source information as well as the trustable list to determine if data sending is allowed. The filesystem I/O overhead is negligible: 0.5 ms for reads and writes.

2) Java Microbenchmark: The CaffeineMark scores are shown in Fig. 4. The scores roughly correspond to the number of Java instructions executed per second. D2Taint and

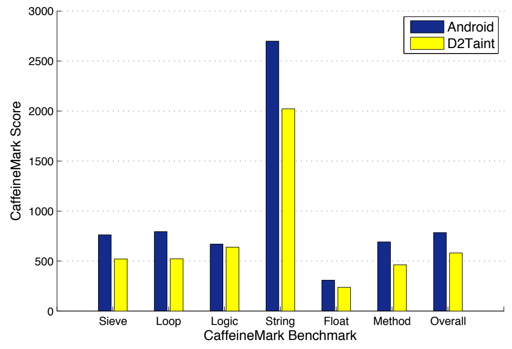


Fig. 4: Microbenchmark: Java overhead

unmodified Android have scores 581 and 784, respectively, so D2Taint's overhead is 25.9%. In contrast, TaintDroid and unmodified Android have scores 967 and 1121, respectively, and TaintDroid's overhead is 14%. Though D2Taint's overhead is higher than TaintDroid's, D2Taint's absolute impaired score (203) does not significantly differ from that of TaintDroid (154). D2Taint's extra overhead is expected since D2Taint needs more operations for the tag system and location information table operations. In contrast, TaintDroid uses only the OR operation to merge tags since its sources are fixed and hard-coded.

We also measured CaffeineMark's memory footprint to determine D2Taint's space overhead. Since the memory footprint value varies with the time after CaffeineMark is launched, we obtained the value immediately after rebooting a system. CaffeineMark consumes 21664 KB and 22528 KB in a unmodified Android system and our D2Taint system, respectively: a 4.0% overhead. This overhead is slightly lower than TaintDroid's, which is 4.4% [4]. Both D2Taint and TaintDroid use the same tag length: 32 bits. The other primary memory used by D2Taint is for the location information table. Note the location information table dynamically increases as more information sources arrive. This overhead is ignored here as CaffeineMark does not access the Internet; hence no entries would be added to the tables.

C. Dynamic Tag System Performance

We evaluate the performance of a dynamic tag system under different situations. To do so, we measure the number of sources recorded in a tag.

In the experiments, we pre-configure four tag schemes for three classes: (1) 2/2/2 hashcode slots for classes 1/2/3 (default); (2) 4/1/1 hashcode slots for classes 1/2/3; (3) 1/4/1 hashcode slots for classes 1/2/3; (4) 1/1/4 hashcode slots for classes 1/2/3. Each hashcode's length is fixed at 5 bits. The tag scheme switching is triggered after 10 new entries are added into the location information tables. We select the "best matched" tag scheme based on the class distributions among these entries.

We write an application to visit the top 100 websites hosted in the U.S. Websites 1–30 are classified into class 1, 31–60 are classified into class 2, 61–90 are classified into class 3, and the remaining 10 are unclassified. The application visits websites 1–100 sequentially or randomly. To emulate tag propagation, we combine several previously downloaded webpages into the final "stolen" data for socket transmission. Thus, the most recent webpage has a higher probability to be selected for combination. We run the experiments 50 times.

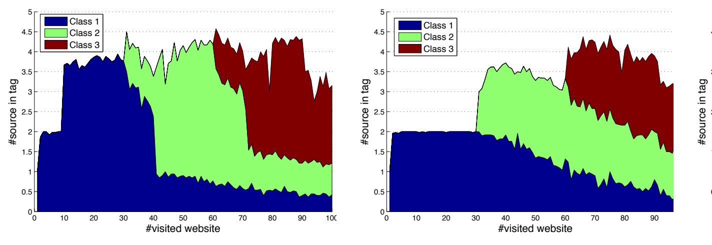
The results are shown in Figs. 5 and 6. For sequential websites, a static tag system can record at most two sources before the class 2 websites are visited. The peak appears after class 3 websites are visited as all 6 available hashcode slots can be used. In our D2Taint, we found tag scheme switching happened at websites 11, 41, and 71. There are some troughs among the class transition period as the tag system has not adjusted yet. Shortly thereafter, the average source number increases to about four sources per tag. The final trough is caused by the unclassified websites as they do not appear in the tag. In general, a dynamic tag system's performance is much better than a static tag system's performance as the former fits the currently visited website classes well.

For random websites, tag scheme switching occurs about five times (on average) with D2Taint. The number of sources remains stable with the number for the static tag system. The dynamic tag system's performance is a bit worse than the static tag system's, as the default tag system is "best" for random websites. Thus, the dynamic tag system is not a good candidate for a totally random situation. But we argue that patterns tend to arise as users visit websites and run applications.

VII. Related Work

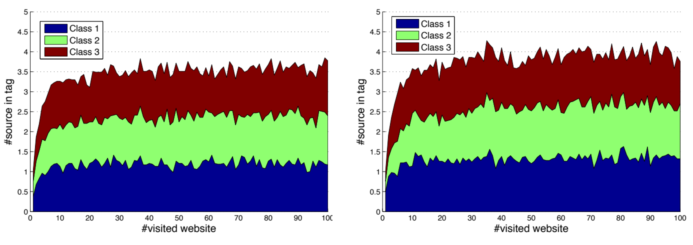
Information leakage in smartphone systems has attracted considerable attention. In numerous instances, third-party applications have leaked personal information to remote servers [2], [21], [22]. Most recently, Carrier IQ [23] has gathered voluminous data from smartphones, apparently without their owners' knowledge. Smartphones' sensors can also leak information such as workers' activity [24] to remote servers. Many systems have been proposed to combat information leakage. TaintDroid [4] tracks information flow from *single* third-party applications. Vision [25] extends TaintDroid to detect implicit information flows. AppFence [26] augments TaintDroid with privacy enforcement mechanisms. Kirin [27] and Saint [28] provide rule-based security mechanisms for Android that restrict application access to sensitive information. SxC [29] adds provable security contracts to Windows Mobile for the same purpose. Other systems [7], [14], [18], [30] leverage static analysis to discern information leaking in Android and iOS applications. D2Taint has two key differences from existing IFT systems for smartphones. First, D2Taint accommodates a large number of sources, including multiple applications, and classifies them into multiple groups. Second, D2Taint's tag structure accommodates varying *types* of sources with dynamic granularity. If many untrusted sources enter the system, each source's hashcode occupies less space, and vice versa.

Most IFT systems store *static* taint tags using shadow memory [4], [31], [32] or tag maps [33]. Usually, each data byte or word corresponds to a byte or word in shadow memory. TaintDroid [4] propagates a static taint tag through the entire Android system; this tag's value does not change. By contrast, D2Taint's tag structure is partitioned into classes, providing



(a) Dynamic (b) Static

Fig. 5: Sequential websites



(a) Dynamic (b) Static

Fig. 6: Random websites

finer granularity than static tags.

More generally, dynamic taint analysis [11], [31]–[34] (also known as "taint tracking") is an approach for information leakage detection. Some dynamic taint analysis approaches are based on whole-system analysis using emulation environments, [32], [35], hardware extensions [9], [12], [17], and per-process tracking with dynamic binary translation [31], [33], [34], [36]. However, these kinds of whole-system analysis are far too heavyweight for resource-constrained smartphones.

VIII. Conclusions

We proposed a novel IFT tagging strategy using differentiated and dynamic tagging. Our strategy partitioned information sources into differentiated classes and stored class and source information in IFT tags. Our strategy enabled dynamic tag structure adaptation in real-time based on received information sources. We designed and implemented D2Taint, an IFT system using our strategy, on real-world smartphones. Our experimental evaluation illustrated D2Taint's potential to detect information leakage with moderate time and space overhead.

References

[1] The Nielsen Co., "Two Thirds of New Mobile Buyers Now Opting For Smartphones," 12 Jul. 2012, http://blog.nielsen.com/nielsenwire/online_mobile/two-thirds-of-new-mobile-buyers-now-opting-for-smartphones/.

[2] S. Thurm and Kane, Y. I., "iPhone and Android Apps Breach Privacy," 17 Dec. 2010, http://online.wsj.com/article/SB10001424052748704-694004576020083703574602.html.

[3] Trend Micro, "Android Malware: How Worried Should You Be?" 16 Jul. 2012, http://blog.trendmicro.com/android-malware-how-worried-should-you-be/.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacymonitoring on smartphones," in *OSDI*, Oct. 2010.

[5] Google Play, http://play.google.com/apps.

[6] Android, http://android.com.

[7] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *POPL*, 1999.

[8] A. C. Myers and B. Liskov, "Protecting Privacy Using the Decentralized Label Model," *ACM Trans. on Softw. Eng. and Methodology*, vol. 9, no. 4, Oct. 2000.

[9] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. August, "RIFLE: An architectural framework for user-centric information-flow security," in *MICRO*, Dec 2004.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," in *SOSP*, 2005.

[11] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *NDSS*, 2005.

[12] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *ASPLOS*, 2004.

[13] D. E. Denning, "A Lattice Model of Secure Information Flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[14] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Commun. ACM*, vol. 20, no. 7, 1977.

[15] N. Heintze and J. G. Riecke, "The SLam Calculus: Programming with Secrecy and Integrity," in *POPL*, 1998, pp. 365–377.

[16] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *USENIX Security*, Aug 2006.

[17] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *MICRO*, 2004.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *USENIX Security*, 2011.

[19] Pendragon Software Corp., "CaffeineMark 3.0," http://www.benchmarkhq.ru/cm30/.

[20] Alexa, http://www.alexa.com/topsites.

[21] D. Moren, "Retrievable iPhone numbers mean potential privacy issues," 29 Sep. 2009, http://www.macworld.com/article/143047/2009/09/phone_hole.html.

[22] C. Davies, "iPhone spyware debated as app library "phones home"," 17 Aug. 2009, http://www.slashgear.com/iphone-spyware-debated-as-app-library-phones-home-1752491/.

[23] J. Brodkin, "Carrier IQ hit with privacy lawsuits as more security researchers weigh in," 2 Dec. 2011, http://arstechnica.com/tech-policy/news/2011/12/carrier-iq-hit-with-privacy-lawsuits-as-more-security-researchers-weigh-in.ars.

[24] M. Fitzpatrick, "Mobile that allows bosses to snoop on staff developed," BBC News, 10 Mar. 2010, http://news.bbc.co.uk/2/hi/technology/8559683.stm.

[25] P. Gilbert, B. G. Chun, L. P. Cox, and J. Jung, "Vision: Automated Security Validation of Mobile Apps at App Markets," in *MCS*, 2011.

[26] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, ""These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications," in *CCS*, 2011.

[27] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *CCS*, 2009.

[28] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *ACSAC*, 2009.

[29] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe, "Security-by-contract on the .NET platform," *Inf. Security Tech. Rep.*, vol. 13, no. 1, pp. 25–32, 2008.

[30] M. Egele, C. Kruegel, E. Kirda, , and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *NDSS*, 2011.

[31] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *IEEE/ACM MICRO*, 2006.

[32] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *ACM CCS*, 2007.

[33] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Privacy Scope: A Precise Information Flow Tracking System for Finding Application Leaks," Dept. of Computer Science, UC Berkeley, Tech. Rep., 2009.

[34] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," in *Int'l. Symp. on Softw. Test. and Anal.*, 2007.

[35] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *USENIX Security*, 2004.

[36] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *ISCC*, 2006.