Owner-centric Protection of Unstructured Data on Smartphones

Yajin Zhou[1]*, Kapil Singh[2], and Xuxian Jiang[1]

[1] North Carolina State University
yajin_zhou@ncsu.edu, jiang@cs.ncsu.edu
[2] IBM T.J. Watson Research Center
kapil@us.ibm.com

Abstract. Modern smartphone apps tend to contain and use vast amounts of data that can be broadly classified as *structured* and *unstructured*. Structured data, such as an user's geolocation, has predefined semantics that can be retrieved by well-defined platform APIs. Unstructured data, on the other hand, relies on the context of the apps to reflect its meaning and value, and is typically provided by the user directly into an app's interface. Recent research has shown that third-party apps are leaking highly-sensitive unstructured data, including user's banking credentials. Unfortunately, none of the current solutions focus on the protection of unstructured data.

In this paper, we propose an owner-centric solution to protect unstructured data on smartphones. Our approach allows the data owners to specify security policies when providing their *unstructured* data to third-party apps. It tracks the flow of information to enforce the owner's policies at strategic exit points. Based on this approach, we design and implement a system, called `DataChest`. We develop several mechanisms to reduce user burden and keep interruption to the minimum, while at the same time preventing the malicious apps from tricking the user. We evaluate our system against a set of real-world malicious apps and a series of synthetic attacks to show that it can successfully prevent the leakage of unstructured data while incurring reasonable performance overhead.

1 Introduction

Smartphone apps tend to contain vast amounts of sensitive data. In many cases, such data items have predefined structure and consistent access semantics across different apps. These data items, such as user's location or contact information, are often regulated by the mobile platform that provides apps access to these items via well-defined platform APIs. For example, apps can retrieve user's geolocation from phone's GPS sensor and subsequently provide location-aware features to users. We classify such data as *structured* data. With the need to protect sensitive information in structured data, a wide variety of security mechanisms

* Part of the work was done when the first author was an intern at IBM T.J. Watson Research Center.

have been developed by the platform developers as well as the research community [19][24][27][30].

At the same time, the apps also alternatively consume data collected directly from the users using the application-controlled visual interfaces (i.e., the users act as *data owners* in this environment). The underlying platform renders minimum to no control in the collection of such data. This type of data, classified as *unstructured* data, relies on the *context* of the app to reflect its meaning and values. One such example is the data collected using an user input box that can have different semantics in different apps. For instance, user can provide his bank credentials in the user input box of one app while he can type the hobbies in another app.

In recent years, there have been several known real-world instances in which apps have leaked unstructured data both *intentionally* and *unintentionally*. In January 2010, several fake banking apps that aim to collect banking credentials were identified [4] in official Android Market. Later in the same year, a fake version of the Netflix app was found to leak user's Netflix credentials to adversary's servers [6]. Besides intentional leaks by these fake (and malicious) apps, genuine apps with vulnerabilities can be exploited to unintentionally leak unstructured data. Recent studies have revealed several instances of benign apps leaking sensitive information, such as user's email login identifier and password [28]. While privacy threat to users is always a major concern, certain information when leaked, can result in serious financial losses.

Unfortunately, all the previous efforts [19][27][30] were focusing on securing structured data. The protection of unstructured data has been largely ignored and left at the mercy of third-party apps. As a result, the owners of unstructured data, i.e., users of third-party apps, have to *blindly* trust these apps and provide security-sensitive data to them. However, the growing number of real-world threats and the sensitive nature of the data that have been leaked emphasize the urgent need for a system-driven solution to protect such unstructured data.

In this work, we are concerned with protecting unstructured data in the presence of *untrusted* third-party apps. To this end, we propose an *owner-centric* approach in which the data owners can determine the security policies for their contributed data. Our solution is based on the insight that data owners could best understand the semantics and sensitivity of their data and hence are best suited to determine who can have access to the data.

Accordingly, we design and implement a system, called `DataChest`, that enables data owners to associate security policies to their data being fed to the untrusted third-party apps. Our system subsequently tracks the flow of the data and applies data owners' policies at strategic exit points in the system. Any policy violation will result in immediate halting of the data transmission.

To encourage real-world acceptability of `DataChest`, we develop several mechanisms to reduce user efforts in specifying policies for user-provided data. In particular, `DataChest` provides persistent policies for *statically* generated input data elements. For *dynamically* generated elements, our system automatically applies the policies based on user's intent. Furthermore, our system provides semantic-aware tag (or policy) suggestions. All these mechanisms reduce user burden and interruption as much as possible, while at the same time guaranteeing that the user's data security is never compromised.

Many of our design choices to reduce user burden have been derived from techniques that have been well proven to be usable in other work streams. For example, the semantic-aware tag suggestion feature based on user's input value (Section 3.4) has been extensively used in search engines, albeit for suggesting related search topics. Other choices are self-intuitive, e.g. if the GUI shown to the user does not change, it is safe to apply previously-specified tags (Section 3.3 and Section 3.5). Usability can be further improved by leveraging additional knowledge, e.g. policies can be pre-specified by trusted authorities such as corporate administrators, and reputation of external entities can be automatically applied using blacklisting databases.

We demonstrate the effectiveness of `DataChest` in tracking unstructured data by analyzing it against popular benign apps from multiple categories. We further evaluate our system against real-world malicious apps and synthetic attacks to show that our system can successfully prevent the unstructured data from being leaked to both (malicious) remote servers and unintended third-party apps. With a CPU-bound benchmark, the results also show that our system has a relative low runtime overhead of 14% with respect to the unmodified Android system. Moreover, the extra time needed to initialize the GUI interface is around 40 ms for a representative real-world scenario, which is a negligible latency that users can actually perceive.

In summary, the paper makes the following contributions:

– To the best our knowledge, we are the first to address the challenge of protecting unstructured data on smartphones. Our system takes an owner-centric approach and engages the data owners to explicitly specify the security policies for their data that are subsequently enforced by our system.
– To minimize user burden, we develop mechanisms to address the challenges of distinguishing both statically- and dynamically-generated input elements, so that they can be effectively tagged with minimum to no user intervention.
– We develop a proof-of-concept system, called `DataChest`, and evaluate its protection capabilities against real-world, malicious mobile apps as well as some synthetically-generated attacks. Our results illustrate that `DataChest` can successfully prevent all such attacks. Our performance results further demonstrate that `DataChest`'s protection mechanism incurs reasonable performance overhead with negligible perceived latency for the end users.

2 Motivating Examples

In contrast to structured data that has well-defined semantics, the semantics of unstructured data can vary substantially in different app contexts. One example of unstructured data is the data entered into input boxes. The exact type of data that users would enter into the boxes cannot be determined without knowing the context of apps. The APIs (`EditText.getText()` in Android platform) used

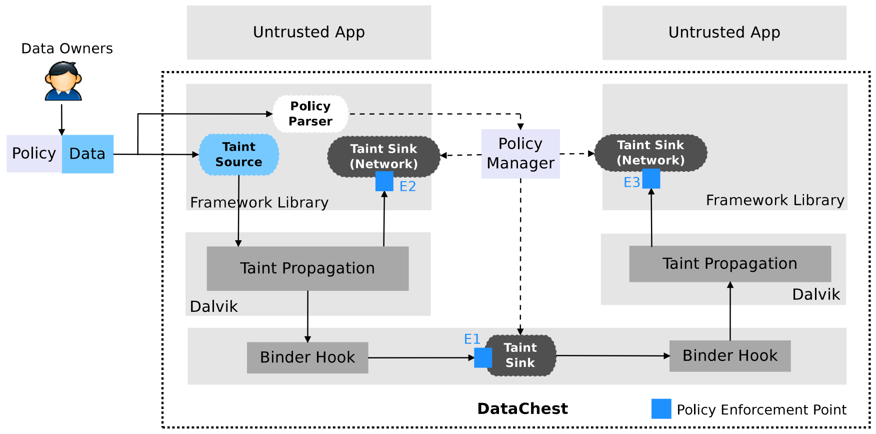


Fig. 1. High-level system architecture of DataChest

to retrieve the values in these input boxes cannot be directly leveraged to understand their semantic meanings. In fact, these values could range anywhere from security-sensitive data such as user's password or SSN to less sensitive data such as user's hobbies.

Since the context of an app is unknown to the underlying Android framework, it does not understand the semantics of any unstructured data entered into the app. As a result, the framework cannot enforce any access control policies (or permissions) that correspond to unstructured data. This further implies that an app can freely access and leak unstructured data without any constraints from the framework. For example, a malicious app can launch phishing attacks by masquerading itself as a banking app [4] and consequently steal users' banking credentials potentially leading to financial loss for the users.

Our work is motivated by real-world threats and aims to provide protection to user-provided unstructured data by means of an owner-centric approach.

3 Design

Figure 1 shows the high-level architecture of our `DataChest` system. When data owners (i.e., the users) provide data to third-party apps, they also include policies that specify how their data can be used and shared with other apps and remote servers. `DataChest` subsequently retrieves these policies and tracks the flow of unstructured data at runtime. Our system will enforce corresponding policies when such data is shared with other apps (E1 in Figure 1) or remote servers (E2 and E3 in Figure 1). In the following, we will describe the associated design challenges and how these challenges are addressed in our system.

3.1 Design Challenges

With an owner-centric approach, our system requires additional efforts from data owners to specify policies. In order to make the system more user friendly and thereby enhance its acceptability, we need to reduce the burden and interruption for users as much as possible, while at the same time, we also need to make sure malicious apps cannot trick users and compromise the security of their data.

Our system leverages TaintDroid [13] to track the information flow of unstructured data. However, TaintDroid only supports limited number of taints (32), which is not sufficient for our system. In our system, we need to track user-provided data for each individual input element. Since these items can significantly vary in number based on the app, we need to support a large number of taints.

3.2 In-context User Policies

Users can specify their security policies in the context of apps, i.e., users specify policies at the time when they actually type their data in the input boxes of a particular app. Since users are entering the data based on their own perception of the app's visual interface, they know the semantics and values of the data they are providing and hence are the best suited to specify the policies based on the context of user inputs. However, the major challenge here is that users need to *explicitly* identify the external entities[3] *before* data items are actually sent out. Users may have no idea of the remote server(s) that are used in the apps and which one should be allowed in advance.

In `DataChest`, we address this challenge by allowing users to specify policies in the context of users' inputs *without* explicitly providing external entities. Specifically, when typing content into user input boxes that may contain sensitive data, users can *tag* the content with meaningful, user-specific, *labels*. This can help users maintain the context of particular user inputs. For example, user can tag the input box that accepts his Paypal password as *Paypal.password*. The flow of tagged information entered into the user input boxes is tracked in the system. When such data is being sent out to remote servers, our system alerts users with the destination and data labels. By showing users the data labels, they can know the types of user inputs that are currently being sent out. Subsequently, users can specify their policies by allowing or disallowing such data transmission temporarily or permanently. For example, they can allow the data with particular label *Paypal.password* to transmit to `paypal.com` permanently while disallowing such data transmission to `evil.com`.

3.3 Persistent User Policies

In order to make our system more user-friendly, we want to reduce user's burden and interruption as much as possible. Note that visual elements, such as user input boxes, can either be statically defined by XML layout files [2] or dynamically generated by apps at runtime in Android. In this section, we will discuss

[3] External entity in our system means remote servers or apps with the different developer's signature.

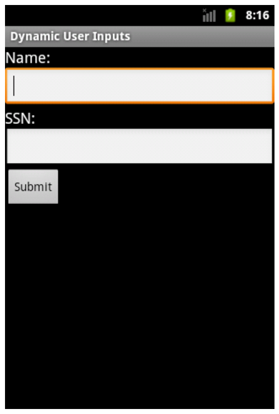
our approach to handle the static user input boxes and leave the dynamically generated ones for Section 3.5.

For static user inputs, we provide persistent tags and policy settings so that users need to tag input boxes and specify the policies only once. Subsequently, for each user input box that has been tagged with labels and associated with policies, the same labels and policies will be applied to this input box automatically every time it is instantiated in the system. A malicious developer might attempt to trick our system by first making the users to enter low-sensitive data into an input box and subsequently tricking the user into entering high-sensitive data into the same input box. However, our system is resistant against such attacks as it compares visual layouts of Android apps (Section 3.5) in order to determine if the input box in question is a mere instance of a previously-tagged box and only applies persistent tags if the visual layout remains same. The malicious developer would need to modify the visual screen shown to the user to make the user enter a different value for the same input box and in such a case, persistent tags would not be applied thus preventing the attack.

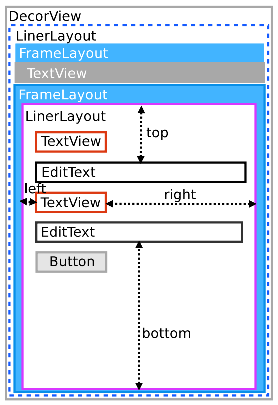
One design consideration is to decide whether we need to maintain the old policies associated with an app when the app is upgraded. From a user-friendly design perspective, we should keep the policies so that users do not need to specify policies again for the app. However, apps are untrusted in our system and *blindly* applying old policies opens up potential avenues for a malicious app to trick the user into giving away sensitive information. For instance, the malicious app can replace an input box with less restricted policies in the older version with an input box that can accept more sensitive data in the new version, thereby enabling the app to leak this sensitive data. In `DataChest`, we take a more restrictive approach and preserve old policies for newly installed app if and only if it has same package name *and* same hash as the older one. We understand that it limits usability in case only minor changes are made between two versions of the app and the semantics of user input boxes are the same. However, it is a trade-off that we made between security and usability. Moreover, the semantic-aware tag suggestion feature described in the next section can make the policy specification for the new app version much easier.

3.4 Semantic-aware Tag Suggestion

We further reduce user burden by providing semantic-aware tag suggestions, i.e., suggestions based on the value of the data being entered. Note that when users tag an input box, they actually correlate the content of their input with the particular label (and its corresponding policies). Therefore, it is possible to infer a user's choice of label for an input box based on the current content of the box. For instance, if the user has tagged an input box with label `SSN` and entered a value of `111-22-3333`, we can suggest the same label to the user when he is entering same value into another input box (in the same app). The suggestion is displayed at the bottom of the input box. The user can accept this suggestion by simply clicking on it and consequently this label (and its corresponding policies) will be automatically applied to the input box.



(a) Screenshot of dynamically generated GUI elements



(b) Visual layout of GUI elements

Fig. 2. Screenshot of an activity and its visual layout

This feature is particularly useful for the case of upgraded version of an app. In previous section, we discussed that the old policies and tags would be discarded after app upgrading. Users can benefit from this feature while using the new version of the app since our system can provide accurate tag suggestions to them based on the content they have provided in the old version.

3.5 Dynamic User Input Elements

Dynamic input elements, such as the input boxes, are created at app runtime and are not predefined in the XML layout file of the app. We cannot assign unique IDs to such dynamically-generated input boxes and consequently have no way to uniquely distinguish them. Therefore, it is not possible to provide persistent tags and policies to such elements. However, requiring users to explicitly tag the user input boxes and specify corresponding policies every time they use the app is a major usability limitation and might not be acceptable. We need to find a better solution to address this challenge.

An effective solution would be to apply tags and policy settings to user input boxes based on their *visual layouts* that are presented to the users, and not solely based on their IDs. For statically-generated user input boxes, their visual layouts are predefined and fixed, hence the IDs of the input boxes effectively reflect their visual layouts. However, it is not the case for dynamically-generated input boxes. If we find a way to compare two visual GUI layouts in one app in different runs, we can automatically apply the same tag settings to them if they are *visually* same.

However, effective comparison of two visual layouts of GUI is challenging and requires an understanding of how GUI elements are created, instanced and how their visual positions are determined. Similar to the DOM objects in browsers, the GUI elements in Android are organized as a tree. All the nodes in this tree are

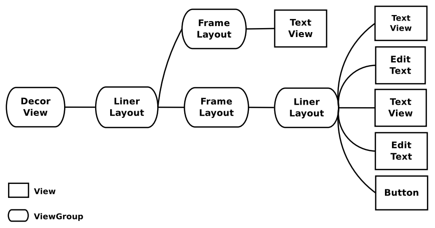


Fig. 3. View tree of the GUI elements in Figure 2

`View` or `ViewGroup` objects [5]. The difference between `View` and `ViewGroup` objects is that `ViewGroup` is the container that can embody other `View` or `ViewGroup` objects. Therefore, all the leaf nodes in this tree are `View` objects while inner nodes are `ViewGroup` objects.

Figure 2 shows the screenshot of an activity with dynamically-generated GUI elements and its visual layout. The corresponding view tree is shown in Figure 3. The root node of the view tree is a special component `DecorView`, which is an internal framework class and represents the top window. It contains a single `LinerLayout` (subclass of `ViewGroup`) object, which has two `FrameLayout` (subclass of `ViewGroup`) children. One of them holds the title of current activity while the other holds the main content of the current activity (another `LinerLayout` object). This `LinerLayout` object contains two `TextView` objects which hold the text "Name:" and "SSN:", two `EditText` objects which can receive user inputs and another `Button` object. For each node in the tree, it knows the *relative* offset to its parent. For example, in Figure 2(b), we can get the *relative* left, right, top and bottom potion of UI elements to its parent (`LinerLayout` object) in dotted lines.

When being initialized, all the objects in the view tree are drawn from the root node to the last leaf node. All the GUI elements are subsequently laid out and positioned on the screen. For each `View` and `ViewGroup` object, it maintains the *relative* position in four dimensions, i.e., left, right, top and bottom, to its parent. Since the view tree represents the visual layout of an activity, we can compare the view trees of two activities to check whether the visual layouts of them are identical. To this end, for each view tree, we generate the corresponding `signature`. If the signatures of two view trees are same, then the visual layouts rendered by these two trees are identical. The algorithm to generated the signature of a view tree is summarized in Algorithm 1. It recursively generates the signature for each sub-tree (inner node) and leaf node and concatenates the generated signature as a string.

One challenge here is how to generate the signature for the leaf nodes, which are the actual UI elements such as `TextView`, `EditText` and `Button`. In our system, we include the properties of a `View` object that can impact its visual layout to generate its signature. For example, we use the actual text values and the four dimensional *relative* positions to its parents to generate the signature for `TextView` object.

Algorithm 1 Signature generation for a view tree

```
1: procedure genTreeSignature($viewTree$)
2:     for all child in $viewTree$ do
3:         genNodeSignature(child)
4:         if (child is inner node) then
5:             genTreeSignature(child)
6:         end if
7:     end for
8: end procedure
```

When users specify policies for dynamically generated user inputs in an activity at first time, our system generates and saves the signature of current view tree along with the specified policies. After that if there is a match between the signature of a new view tree with a saved one, our system will automatically apply the saved policies to the dynamic user inputs in the new view tree.

Our technique to generate and compare view tree signatures ensures that if two view trees are same, their corresponding visual layouts are identical. However, there are instances in which visual layouts could be same even when the view trees are not identical. For example, two `TextView` objects with text value "Na" and "me" placed next to each other on the screen may have same visual layout as that of one `TextView` with text value "Name". In such scenarios, we cannot automatically apply the tag settings even the visual layouts of two activities are same. Our approach is conservative in such cases as we err on the side of security by not allowing malicious apps to trick the users. At the same time, we can still reduce user burden when using benign apps.

4 Implementation

We have implemented a working prototype of `DataChest` by extending TaintDroid [13] (based on Android 2.3.4[4]) and Android framework in several significant ways. In this section, we illustrate the details of our system implementation.

4.1 User-provided Unstructured Data

As discussed in Section 3.2, users associate security policies with user inputs within the app context by first tagging user input boxes with custom, user-defined labels that reflect the semantics of the input boxes. In our current implementation, we supplement the default user input method (i.e. the on-screen keyboard) with a special *tag* button to allow users to provide labels to the input boxes. We believe that it is a convenient approach for the users as they can enter values and their corresponding labels from a single UI (input method). However, the default UI cannot be leveraged for providing labels if the app uses its own

[4] The latest version of TaintDroid is based on Android 4.3. We leave the porting of our prototype to this version of TaintDroid as our future work.

custom input method. In such a case, our system provides an alternate way to enter labels using a UI that is triggered when a user keeps his finger focused on the input box for a relatively longer period.

To support semantic-aware tag suggestions for reducing user burden, we save the mapping between the data label and the hash value of the content entered by the user into the input box. Note that we do not save the user's input as plaintext for privacy concerns. To this end, we monitor the content of the *tagged* user input boxes by hooking `onTextChanged()` method and update the hash value accordingly. For the *untagged* user input boxes, we also hook the `onTextChanged()` method to compare the hash value of currently typed content with the saved ones. If there is a match, we display the corresponding saved data label under the current input box as a suggestion to users.

For dynamically-generated user inputs in one app, we automatically apply the tag settings if their visual layouts do not change in different runs. When users tag the dynamic user inputs, we save the current view tree and the tag settings, i.e., data labels for this view tree. The saved data labels will be automatically applied to a new activity (in the same app) if its view tree is identical to a saved one. For this purpose, we generate the signature for the view trees by recursively generating the signatures of both *inner* nodes and *leaf* nodes in the view tree. In the Android platform, function `ViewGroup.performTraversals()` is called when current GUI is drawn or redrawn. We hook into this function to generate the signature for the whole view tree.

4.2 System-wide Information Tracking

4.2.1 Taint Tag Format Our system needs to track the flow of user-provided data. For this purpose, we extend TaintDroid [13] in our system. Note that our system design is not restricted to only TaintDroid and we can readily leverage other information tracking systems if such systems are available in the future. In the following we leverage TaintDroid as an example to describe information flow tracking in our system.

As previously stated, one major challenge of using TaintDroid to track information flow is that it only supports a limited number of taints. Specifically, it encodes the taints into a 32-bit tag, in which each bit denotes a taint. However, in our system, any user input box represents a different taint and needs to be tracked independently. That is because during program execution, the data from different sources (different user input boxes from different apps for example) can be combined together and we need to know the exact source of the data (e.g., from which user input box in which app) and check the policies when combined user inputs are being sent out.

In `DataChest`, we extend the format of TaintDroid's original taint tag to support large number of taints. Figure 4 shows the format of taint tag used in our system. Instead of directly using the 32-bit taint tag to place taints, we use a linked list to store the actual taint tags. To distinguish from the original taint tags, we call the actual taint tag as *policy tag* in our system. For each policy tag, we need to store the source information of data that our system is tracking.

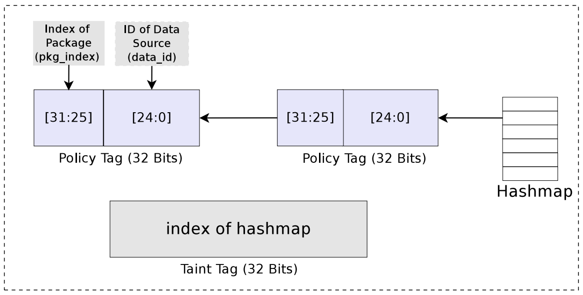


Fig. 4. The taint tag structure in DataChest

Such source information is denoted as the identification of data source, such as the ID of a user input box, and this identification is directly encoded into the policy tag (`data_id` field).

4.2.2 Taint Propagation During program execution, the taints will be propagated through the whole system. This makes sure that even the tainted data is converted to other format, it is still being tracked. TaintDroid propagates taints by extending the Dalvik virtual machine and applying taint propagation rules.

Because of the differences between the format of taint tags, the original rules used to propagate taints need to be changed accordingly. Specifically, for the operation of `result := data`$_1$ `+ data`$_2$, TaintDroid can directly combine two tags together using `or` operation (`tag(result) = or(tag(data`$_1$`), tag(data`$_2$`))`). However, in our system, we have to merge two linked lists of policy tags together and place the address of new linked list into the hash map. Finally, the index of new linked list in hash map is encoded into the taint tag.

we also extended TaintDroid to add other features that were required for our implementation. One such example scenario is that TaintDroid propagates taints in native methods of system libraries. In particular, TaintDroid uses a method profile (a list of (from, to) pairs) to indicate information flow between variables, which may be method parameters, class variables, or return values in the *same* native method. However, this method profile may miss the information flow that crosses *different* native methods *without* any use of a variable. On such example is the `Md5MessageDigest` class in which the taint should propagate from the parameters of a void function `void update(byte[] input)` to the return value of `byte[] digest()` that has no parameters. The original method profile cannot handle this. We extend the method profile that can propagate taints across *different* methods *without* using variables.

4.2.3 Taint Sources and Sinks For user-provided unstructured data, when one input box is tagged with a particular label, our system treats this input box as a taint source and creates corresponding taint tags (and policy tags) for the data retrieved from this input box.

When the data with taint tags is reaching certain exit points, our system checks and enforces the corresponding policies. In our system, we treat such

Table 1. List of apps that leak user-provided unstructured data

App/Malware Name	Malware?	Type of Unstructured Data
FakeNetflix	Y	Netflix Login Identification and Password
Repackaged Paypal	Y	Paypal Username and Password
Youdao Dictionary	N	Netease Username and Password

exit points as taint sinks. Similar to TaintDroid, the network interface is one taint sink in our system. If no policy has been specified, we block the current network operation (using Linux pipe) and display a popup window (through a management system app) to let users make decision. Besides network interface, there is another new taint sink in our system. That is the point where one app is sharing the data with another app through the binder interface. By checking and enforcing policies (see E1, E2 and E3 in Figure 1) at taint sinks, our system can prevent the app from sharing (tainted) unstructured data with unauthorized external entities.

5 Evaluation

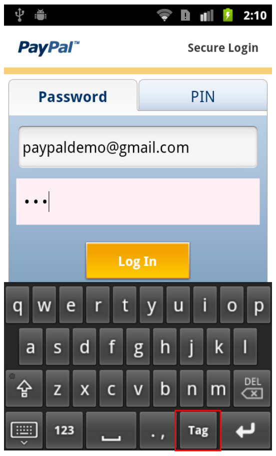
In this section, we present our evaluation of the effectiveness and performance overhead of `DataChest`.

5.1 Effectiveness

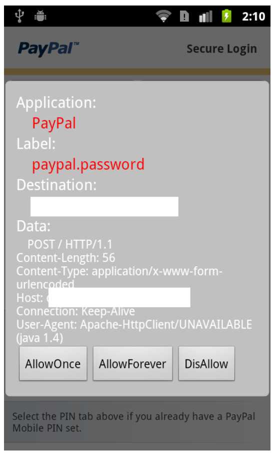
To demonstrate the effectiveness of `DataChest`, we downloaded 50 popular benign apps from Google Play and subsequently selected 23 apps that collect sensitive user-provided unstructured data, such as login credentials. We use these apps to evaluate the effectiveness of our system in tracking the information flow of sensitive user-provided unstructured data.

Our evaluation shows that all of the user-provided unstructured data to these apps can be successfully tracked by our system. Note that even in cases where the app does not use the default `EditText` class to accept user inputs, our system still can successfully track the unstructured data provided by users. One such example is the search input box in `Amazon Mobile` app that implements its own class (`SearchEditText`) to accept user inputs. This class extends the default `EditText` class (which is a subclass of the base GUI class `View`) to include some app-specific features. Our system mainly hooks the functions in the base framework GUI class (`View`) and all other classes that extend from this class automatically inherit these hooked functions. If these hooked functions in `View` class are overwritten in subclasses and not called from the subclasses, the GUI will not be successfully initialized.

To demonstrate the capability that our system can prevent user-provided sensitive unstructured data from being leaked by malicious apps, we evaluated it against two malware samples that leak sensitive data to remote servers. The



(a) Tag button to tag user input boxes



Application:
PayPal
Label:
paypal.password
Destination:
[redacted]
Data:
POST / HTTP/1.1
Content-Length: 56
Content-Type: application/x-www-form-urlencoded
Host: [redacted]
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

AllowOnce
AllowForever
DisAllow
Select the PIN tab above if you already have a PayPal Mobile PIN set.

(b) Popup to let users make runtime access decision

Fig. 5. Our system prevents data leakage from a repackaged Paypal app

first app is the `FakeNetflix` [6] malware that was discovered in October 2011. It disguises itself as the real Netflix app and leaks the user's Netflix credentials to a remote server. This malware only masquerades the UI of the Netflix app and does not provide any real functionality of video streaming. Another app is a repackaged Paypal app that we developed in-house for our evaluation. In contrast to the previous app, this repackaged Paypal app is more stealthy since it has the same functionalities as the real Paypal app. However, in the background, it leaks the user's Paypal username and password to a remote server.

Moreover, besides intentional leakage of unstructured data by malicious apps, some benign apps have been found to be vulnerable and can be exploited to leak unstructured data. We also use one such app, `Youdao Dictionary` [7], for our evaluation. This app has an open content provider that stores the username and password of a Netease account in plain text [28]. Any malicious app on the phone can access the stored account information through the open content provider. The apps and the corresponding types of unstructured data that could be leaked are shown in Table 1.

Our experiments show that `DataChest` can successfully prevent data leaks by these malicious and vulnerable apps. In particular, when using these apps, we tag the input boxes that accept user's credential with specific labels (`paypal.password` for example). Figure 5(a) shows our enhancements to the default on-screen keyboard to include a special tag button to tag user input boxes for the repackaged Paypal app. The flow of data from these tagged user inputs will be tracked in the whole system. As a result, when such tagged user inputs are being leaked to remote servers, the user would be notified using a pop up notification. This prompt includes the destination of this data transmission and the data label that provides the semantics of the data. Users can make decisions to allow or

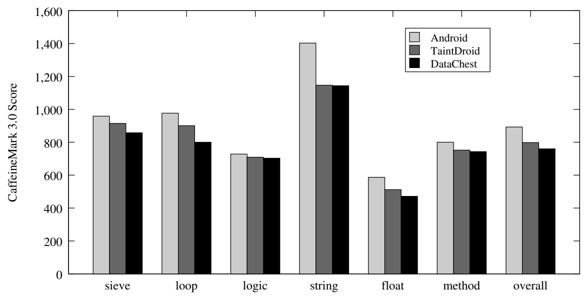


Fig. 6. Evaluation results from the Caffeine benchmark

block this data transmission temporally or permanently. Figure 5(b) shows this pop up window. Note that users only need to tag input boxes and make their decision only once since all the tagged user inputs (with their data labels) and users' decisions (or policies) will be saved (see Section 3.3).

5.2 Performance Overhead

In this section, we study `DataChest`'s performance overhead. All the evaluations are performed on Google Nexus S running Android 2.3.4 that is modified for `DataChest`.

5.2.1 Dalvik Microbenchmark `DataChest` extends TaintDroid's internal taint tag format and taint propagation logic in Dalvik virtual machine. Therefore, we want to study the performance overhead introduced by this extension. To this end, we used an Android port of the standard CaffeineMark 3.0 [1] benchmark and reported the scores of this benchmark running on original Android, TaintDroid and our system in Figure 6. The x-axis shows the different operations performed by this benchmark and y-axis shows the corresponding score of each operation. These scores are useful for relative comparisons.

The benchmark results are consistent with the results reported in TaintDroid. The String operations of both TaintDroid and DataChest have higher performance overhead than arithmetic and logic operations due to the additional memory comparisons [13]. The overall score of Android is 893 while it is 798 and 760 for TaintDroid and DataChest, respectively. It basically implies that DataChest has a 14% overhead with respect to the unmodified Android system and 5% overhead with respect to TaintDroid.

5.2.2 GUI Microbenchmark To reduce the user burden in `DataChest`, we generate the signature of a view tree and use this signature for visual layout comparison (see section 4.1). In this section, we evaluate the performance overhead due to this signature generation and comparison. For this purpose, we developed a testing app which dynamically creates 40 `TextView` objects and 40 `EditText` objects in an activity. Additionally, we used a real library (Paypal mobile payment Library [3]) which dynamically generates its UI. We calculate the time

Table 2. The evaluation results of GUI Microbenchmark (time is in milliseconds)

	#1	#2	#3	#4	#5	average
Signature generation (Our testing app)	99	93	76	133	139	108
Signature generation (Paypal payment library)	63	65	29	13	32	40
Signature comparison	0	0	0	0	0	0
Policy retrieval	2	3	3	1	2	2
Policy insertion	4	9	5	6	7	6

used to generate the signature for the view tree and the time to compare two view trees. For each evaluation, we measure 5 times and report the results in Table 2. We find that the time used to generate signature of a view tree is around 100 ms for our testing app and 40 ms for Paypal payment library. Note that our testing app has 80 dynamic GUI elements that is considerably more than the number of GUI elements of a typical app. Surprisingly, the time used to compare two signatures is nearly zero. This is because the comparison is merely a string comparison. Moreover, in exiting points, our system needs to retrieve the policies from another separate app which is responsible for maintaining policies. We also evaluate the time latency that was introduced by this operation. The experiments show that the time used to retrieve and insert a policy is 2 ms and 6 ms, respectively. We believe that the time latency introduced by our system is negligible that users can actually perceive and is within the acceptable range of 50–150ms [25].

6 Discussion

In this section, we discuss the current limitations of our system and propose possible solutions.

Although the system is effective in preventing data leaks, it might be limited against certain advanced attacks. One possible attack would be the side-channel attack. For example, instead of getting the contents of user input boxes and then sending them to the remote server, the attackers continuously take a screenshot of the current activity and send these screenshots to the remote server. The current solution cannot handle such side-channel leaks. Another possible attack is the man-in-middle attack to steal data if the data is being sent out without any encryption. In our work, we do not consider such attack and trust the network infrastructure.

The current policies in our system use the host name as an identifier for external entity, which may be ineffective for a proxied network connection. In this case, all the connections will go to this proxy first, not the real remote servers. In the presence of a proxy, users have no idea of the real destination of the data transmission and hence cannot make an informed decision. Users may have similar situation in the case of remote servers without meaningful domain names. Nevertheless, users can still protect their data by blocking all transmissions of the tagged user inputs, though this may break the legitimate functionalities of some benign apps.

For user-provided data, our system requires users to specify the policies. However there are some potential ways to reduce user's burden when specifying policies. For instance, one user's policies of one app could be shared online with other users so that other users do not need to specify the policies for that app again. In the scenario of BYOD, all the policies could be specified and pushed by enterprise device management platform, instead of users. From the perspective of usability, our system provides several ways to reduce user burden (Section 3.3, Section 3.4 and Section 3.5). We believe that there is still potential room for improvement by better understanding the system's usability (e.g. via user studies) and we plan to explore this as part of our future work.

Since we extend TaintDroid to track the flow of unstructured data, our system is also limited by some of TaintDroid's inherent limitations. First of all, TaintDroid only tracks data flows (i.e., explicit flows) and does not track control flows (i.e., implicit flows) as demonstrated in ScrubDroid [23]. As a result, it is possible that malicious app may use implicit flow to leak unstructured data to remote servers. To solve this problem, static analysis may be deployed to analyze the apps. Secondly, the taint propagation between different apps and files is still coarse-grained. For example, the whole IPC message between different apps shares one taint tag, which may cause data to be over tainted. Thirdly, TaintDroid does not support native code in third-party apps. To prevent malicious apps from using native code to leak unstructured data, we do not execute the third-party apps with native libs and thus may cause compatibility issues. Fortunately, the number of apps with native code is only around 5% [29]. We leave the work of extending TaintDroid to provide information flow tracking in native code as our future work. As mentioned before, our system design is not bound to TaintDroid and we can leverage advanced information tracking system on Android if such system is available in the future.

7 Related Work

Smartphone privacy issues have attracted a lot of interest in recent times. Previous research works reveal that third-party apps [12, 13] along with in-app advertisement libraries [15] are actively leaking user's private information. To deal with this problem, researchers propose solutions to provide fine-grained control of private information on smartphones. Such solutions include TISSA [30], Apex [21], Aurasium [27], AdDroid [22] and AdSplit [24]. Our work has a different focus from these works since their main focus is to protect structured data while our work is dedicated to the protection of unstructured data.

At the same time, the classic confused-deputy [18] problem or capability leaks are identified on Android. Examples include ComDroid [10], CHEX [20] and Woodpecker [16]. They employ static analysis to identify such problems in third-party apps and pre-loaded apps. Accordingly, possible solutions [8][11][14] are proposed to mitigate such threats. Our work does not intend to detect such problems. But our system can be used to prevent the unintentional data leak by the apps with this problem.

Among the most related ones, AppFence [19] leverages TaintDroid to track information flow and protects private data from being leaked. However, AppFence's focus is on structured data, while our system is protecting unstructured data. TaintEraser [31] shares a similar design to prevent unwanted information exposure, including user inputs. Our system provides more fine-grained polices to let data owners specify the external entities that the data can be shared with. D2Taint [17] expands TaintDroid to track the information coming from Internet sources. Our system tracks and protects unstructured data coming from users.

A recent system called DataSafe [9] allows data owners to specify particular policies to protect their data. Our system has several key differences from DataSafe. First, the target platforms are different. Our system is concerned with the protection of unstructured data on smartphones while DataSafe aims to protect sensitive data on desktop or cloud computing servers. Second, our system addresses the challenge of user-friendly policy specification and provides several mechanisms to reduce user burden, which is critical for smartphone platforms, while DataSafe does not address this challenge. Third, DataSafe is based on hardware-assisted information flow tracking while ours is software-based tracking. The requirement of custom hardware is a challenge for deployment. Another system CleanOS [26] evicts the sensitive data such as user-provided password, from the phone and keeps a clean environment all the time. Our system instead ensures that such sensitive data cannot be obtained by unintended (and potentially malicious) external entities.

8 Conclusions

We presented the design of a system, called `DataChest`, that offers protection of unstructured data in the presence of untrusted third-party apps. Our system develops an owner-centric approach in which data owners (i.e., users) can determine the security policies of their contributed data. We enhance the usability of the system by developing several mechanisms to reduce user burden, while ensuring the security of the system is never compromised. Our evaluation shows `DataChest` is effective in preventing leakage of unstructured data against a variety of attacks and incurs reasonable performance overhead.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855036 and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/.
[2] Layouts. http://developer.android.com/guide/topics/ui/declaring-layout.html.
[3] Mobile Payment Libraries. https://www.x.com/developers/paypal/products/mobile-payment-libraries.
[4] Rogue phishing app smuggled onto Android Marketplace. http://www.theregister.co.uk/2010/01/11/android_phishing_app/.
[5] UI Overview. http://developer.android.com/guide/topics/ui/overview.html.
[6] Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna? http://www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna.
[7] YouDao Dictionary. https://play.google.com/store/apps/details?id=com.youdao.dict.
[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, 2012.
[9] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A Software-Hardware Architecture for Self-Protecting Data. In *CCS*, 2012.
[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011.
[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, 2011.
[12] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.
[13] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Symposium on OSDI*, 2010.
[14] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.
[15] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM WiSec*, 2012.
[16] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.
[17] B. Gu, X. Li, G. Li, A. C. Champion, Z. Chen, F. Qin, and D. Xuan. D2Taint: Differentiated and Dynamic Information Flow Tracking on Smartphones for Numerous Data Sources. In *INFOCOM*, 2013.
[18] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22, October 1998.
[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For": Retroffiting Android to Protect Data from Imperious Applications. In *ACM CCS*, 2011.
[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS*, 2012.
[21] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *ASIACCS*, 2010.
[22] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ASIACCS*, 2012.
[23] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafarn. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *SECRYPT*, 2013.
[24] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, 2012.
[25] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3^{rd} edition, 1998.
[26] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *USENIX Symposium on OSDI*, 2012.
[27] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, 2012.
[28] Y. Zhou and X. Jian. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS*, 2013.
[29] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.
[30] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Trust*, 2011.
[31] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. In *ACM Operating Systems Review*, 2011.