✓ Artifact evaluated by FSE

Detecting Sensitive Data Disclosure via Bi-directional Text Correlation Analysis

Jianjun Huang Xiangyu Zhang
Department of Computer Science
Purdue University, USA
{huang427, xyzhang}@cs.purdue.edu

Lin Tan
Electrical and Computer Engineering
University of Waterloo, Canada
lintan@uwaterloo.ca

ABSTRACT

Traditional sensitive data disclosure analysis faces two challenges: to identify sensitive data that is not generated by specific API calls, and to report the potential disclosures when the disclosed data is recognized as sensitive only after the sink operations. We address these issues by developing BIDTEXT, a novel static technique to detect sensitive data disclosures. BIDTEXT formulates the problem as a type system, in which variables are typed with the text labels that they encounter (e.g., during key-value pair operations). The type system features a novel bi-directional propagation technique that propagates the variable label sets through forward and backward data-flow. A data disclosure is reported if a parameter at a sink point is typed with a sensitive text label. BIDTEXT is evaluated on 10,000 Android apps. It reports 4,406 apps that have sensitive data disclosures, with 4,263 apps having log based disclosures and 1,688 having disclosures due to other sinks such as HTTP requests. Existing techniques can only report 64.0% of what BIDTEXT reports. And manual inspection shows that the false positive rate for BIDTEXT is 10%.

CCS Concepts

•Software and its engineering → Software testing and debugging; •Security and privacy → Software security engineering;

Keywords

Sensitive Data Disclosure, Bi-directional Text Correlation, Android apps

1. INTRODUCTION

Sensitive data disclosure has been a long-standing challenge for data security. By accessing the disclosed sensitive information, adversaries can learn about the system and then conduct attack [28, 25]. A prominent example is the OpenSSL Heartbleed vulnerability disclosed in 2014. The OpenSSL versions with such a flaw allow remote attackers to retrieve sensitive data, for example, user authentication credentials and secret keys [12, 38]. Attackers can then compromise the target systems with the disclosed sensitive information.

The proliferation of mobile devices [13, 33] makes the situation even worse since mobile devices process a lot of sensitive user data. Previous studies showed that it is common that mobile apps undesirably disclose sensitive user information [26, 39, 10]. Many techniques have been proposed that work at the system level or the application level, static or dynamic [21, 9, 14, 8]. Haris *et al.* provide a comprehensive list of the approaches to detecting sensitive information disclosures in mobile computing [16]. All these approaches require definition of the sensitive data sources, usually certain APIs whose return value is sensitive. With the definition, if *forward* data flow is observed between taint sources and sinks, disclosure defects are reported. Later, researchers realized that some generic APIs may return sensitive values, depending on the context, although they may return insensitive values in many cases. SUPOR [17] and UIPicker [27] aimed to identify which user inputs on the user interfaces can be sensitive. Then the sensitive inputs are associated with the variables in the code such that static or dynamic forward data flow analysis can be applied to detect the potential sensitive user inputs disclosures. Sensitive user inputs are identified in the context of the user interfaces which contain text or graphical information to instruct what the users should enter.

However, the above solutions still have limitations. Sensitive data may come from generic API methods not related to UI (*e.g.,* loading data from some file or receiving data from network). In these cases, most existing approaches would not work properly. We cannot simply treat the generic APIs as the taint sources as that will lead to a large number of false warnings. In addition, forward data flow analysis is insufficient. In many cases, a piece of data may be first emitted through a sink and then later typed as sensitive. There may not be any forward data flow from the type revelation point to the sink point.

In this paper, we develop BIDTEXT, a technique to detect data disclosures by examining the text labels correlated with variables. The text labels, either from the code (*e.g.,* the textual keys in key-value pairs) or the UI, provide rich information about the data contained in the variables. BIDTEXT extracts these labels, and leverages a novel type system to propagate these labels through both backward and forward data flow. Data disclosures are reported when a parameter at a sink point is typed with a sensitive textual label. The bi-directional propagation scheme is unique and different from the traditional unification based type inference systems. It features the capability of avoiding undesirable unification of text labels, enabling a low false positive rate. Backward propagation allows BIDTEXT to capture cases in which data sensitiveness is revealed after the data is sent through some sink.

Our work makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950348

```
 1 class CampaignActivity_20 implements Handler.Callback{
 2   CampaignActivity act;
 3   CampaignActivity_20(CampaignActivity a){
 4     this.act = a;
 5   }
 6   public boolean handleMessage(Message msg){
 7     Bundle b = msg.getData();
 8     String dt = b.getString("data");
 9     Log.d("CampaignActivity", "Got data back: " +
          dt);//sink
10     Runnable r = new CampaignActivity_20_1(dt);
11     act.runOnUiThread(r);
12     return false;
13   }
14 }
15 class CampaignActivity_20_1 implements Runnable{
16   String jsonString;
17   CampaignActivity_20_1(String data){
18     jsonString = data;
19   }
20   public void run(){
21     JSONArray jsonArray = new JSONArray(jsonString);
22     int len = jsonArray.length();
23     for (int i=0; i<len; i++) {
24       JSONObject json = jsonArray.getJSONObject(i);
25       String url = json.getString("avatar_url");
26       ImageView iv = ... // omitted
27       displayImage(url, iv); // omitted
28       String un = "<b>" + json.getString("username") +
            "</b>" + json.getString("created_at");
29       TextView tv = ... // omitted
30       tv.setText(Html.formHtml(un));
31       String c = json.getString("content");
32       TextView ctv = ... // omitted
33       ctv.setText(Html.fromHtml(c));
34       // ...
35     }
36   }
37 }
```

Figure 1: Motivating example from app *com.buycott.android*.

- We propose BIDTEXT, a novel method to detect sensitive data disclosures. BIDTEXT leverages constant text labels and features a novel type system that performs bi-directional text label propagation.
- We implement a prototype of BIDTEXT for Android apps, and evaluate it on 10,000 apps. BIDTEXT reports 4,406 apps that have sensitive data disclosures, with 4,263 apps having log based disclosures and 1,688 having disclosures due to other sinks such as HTTP requests. Existing techniques [7, 17] can only report 64.0% of what BIDTEXT reports. And manual inspection shows that the false positive rate for BIDTEXT is 10%.
- BIDTEXT is available at https://bitbucket.org/hjjandy/toydroid.bidtext.

2. MOTIVATING EXAMPLE

We use a real-world Android app *com.buycott.android* to motivate our technique. It is an app that allows users to check the company/vendor of a product by scanning the product's barcode. It even allows users to view the family tree of the company/vendor. Users can then make decision on whether this is a company that rips off its customers so that they do not want to have business with. Users can also start/join campaigns against specific companies [1].

Fig. 1 shows a piece of simplified code snippet from the app. The app sends a request to the Web server and obtains a list of post messages. The HTTP response is converted to a string in the app and then sent to a handler via a `Message` object. The following operations are present in the code snippet. At line 7, a key-value mapping

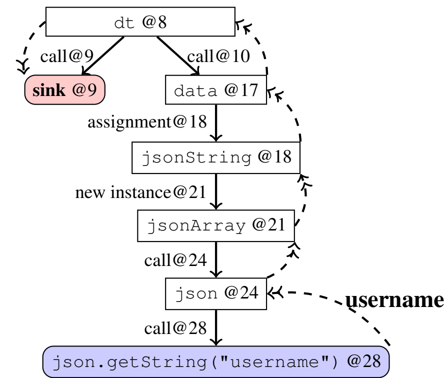


Figure 2: Data flow (solid arrows) and type propagation (dashed arrows) for Fig. 1.

is retrieved from the `Message` object. Then the data string of the message is obtained from the mapping at line 8. Right after that, the data string is written to the log file at line 9. Note that writing to a log file is usually considered as a sink for data disclosures [7, 17, 22] because log files can be accessed by malware[1]. After the logging operation, the app instantiates a `Runnable` object with the data string at line 10, which runs in the UI thread (line 11) to allow interactions with UI elements.

The data string is transmitted to the `Runnable` instance via the instantiation at line 10. Inside the constructor at line 17, the data is stored in a field variable `jsonString` at line 18. When the UI thread is running, the `run()` method at line 20 is invoked. The data string is converted to a `JSONArray` object at line 21 which is then iterated. Every element in the array is a `JSONObject` (line 24). The app then obtains the URL for the avatar image, the corresponding user Id, the time of creation and the content of the post message by looking for the values via corresponding keys in the JSON object (lines 25, 28, and 31). All such information is shown on some UI elements (*e.g.,* line 33).

Now let's consider the potential sensitive data disclosure in this running example. Based on the above description, the data falling into the sink at line 9 comes from the Web server. We later know that the data contains some sensitive user account information. In other words, the app retrieves the sensitive user account information from the server and writes it to the local log file without any encryption. This is a typical kind of undesirable information disclosure [24, 40] that emits sensitive information from server such as user account, balance in bank account, and employee salary to local files.

Traditional sensitive data disclosure analysis inspects the data flow between some sensitive source point, for example, an API call whose return value can be easily recognized as sensitive (*e.g.,* `TelephonyManager.getDeviceId()` in Android), and a sink point (*e.g.,* a file write or a socket send). If forward data flow can be discovered from the source point to the sink point, a disclosure problem is reported. In this example, while we do have data flow from the Web server response to the logging operation but we cannot determine whether the response contains sensitive data *from the operations along the data flow*. If we treat all data from server sensitive, a lot of false alarms will be produced; but if we simply ignore them, we miss true disclosures as in this example.

Different from the traditional disclosure analysis, our technique

[1]The recent version of Android has substantially mitigated this problem by limiting access to log files. But there are still a large number of devices running old versions of Android. Note that BIDTEXT is general to support various configurations of sink points.

relies on the observation that the sensitiveness of data used in applications can be recognized through examining the textual information involved in the operations. Such texts are constant strings in either the code or the user interfaces. We randomly sampled 2,000 Android apps and found that on average each app contains 76.7 constant strings in layout files (*i.e.,* XML files used to statically define UIs) and 151 constant strings in app code. These constant strings often provide rich information about what is being held by the corresponding variables. For example, in Fig. 1, method call `json.getString("username")` at line 28 uses a constant string "*username*". We can infer that the JSON object contains some sensitive user Id. Since the JSON object is part of the Web server response, according to the work flow, we can conclude that the response contains sensitive information. Thus the logging operation at line 9 should be reported as a sensitive data disclosure.

Note that even if we recognized that the JSON object at line 28 contains sensitive information, we could not detect the disclosure problem using traditional analysis techniques that try to find forward data flow from source points to sink points. We show the data flow via solid arrows in Fig. 2, starting from retrieving the data from the key-value mapping (line 8). If we treat line 28 as a source point, we cannot get a forward data flow path from the source point to the sink point. Thus the disclosure defect is still missed after we augment traditional techniques with our new sensitive data recognition method.

BIDTEXT solves the problem by introducing *bi-directional* propagation. Instead of propagating tags like *tainted* and *untainted* in traditional techniques, our approach uses the constant strings as the tags and propagates both backward and forward. As the dashed arrows in Fig. 2 show, constant text "*username*" is propagated backward from the method call at line 28 to the variable `json` created at line 24, and so on. Consequently, variables `jsonArray`, `jsonString`, `data` and finally `dt` are tagged with the text "*username*". Intuitively, it means all these variables contain sensitive user Id information. Next we forwardly propagate the tag from line 8 to the sink point at line 9. Therefore, the logging statement operates on variables that are associated with text "*username*". By applying this approach to the whole code snippet, we obtain the set of correlated text as {"*CampaignActivity*", "*Got data back:*", "*data*", "*avatar_url*", "*username*", "*created_at*", "*content*"}. The first two textual tags are associated to the variable directly at the sink point. Tag "*data*" is propagated to the variable (at the logging statement) in a forward manner. The remaining texts are propagated to the sink point via a bi-directional manner discussed above.

BIDTEXT also associates UI texts to variables. UI often contains texts that also indicate the sensitiveness of data shown on the UI (see [17, 27]). We examine the corresponding layout file to get the texts, add them to the tag set of the related variables and propagate them like the texts found in the code. In the example, we can find several code locations that interact with the UI (*e.g.,* line 33), through which we identify the corresponding layout files to collect UI texts. However, the content of the UI is dynamically created and none of the UI elements holds constant texts. Therefore, no GUI texts are propagated to the sink point in this example.

Next we apply a natural language processing (NLP) technique to the tag set of the sink point to find out if the texts can tell the sensitiveness of the variable `dt`. Among the collected texts, "*username*" matches a predefined sensitive keyword. Thus our technique reports a sensitive data disclosure problem for the logging operation at line 9.

3. DESIGN

We propose BIDTEXT, a static bi-directional text correlation analysis approach, to detect sensitive data disclosures. BIDTEXT combines both the bi-directional propagation and the new approach that uses internal constant texts to identify sensitive variables as illustrated in Section 2.

Program	p ::=	s*	
Statement	s ::=	v := t	/*constant string in code*/
	|	v := i	/*UI-related Id*/
	|	v := c	/*values of other types*/
	|	$v := \ominus v_1$	/*unary assignment*/
	|	$v := v_1 \oplus v_2$	/*binary assignment*/
	|	call(m,$v_a \rightarrow v_f$)	/*v_a/v_f actual/formal arg*/
	|	v := return(m,v_r)	/*m returns v_r to v*/
	|	v := apicall(m,v_a)	/*API call to method `m`*/
	|	IF(v) {s_t} ELSE {s_f}	
	|	LOOP {s}	/*loop structure*/
	|	$v := \phi(v_t, v_f)$	/*value merging in SSA*/
Variable	v		
Method	m		
String	t		
ID	i		
Value	c		/*Non-str, non-Id Values*/

Figure 3: Language.

3.1 Language Abstraction

To simplify our discussion, we introduce an abstract language. The language is presented in Fig. 3. We only model the language features that are related to explaining the text correlation analysis and the bi-directional propagation. Others are abstracted away or simplified. As we discussed in Section 2, we leverage the constant texts in the code as well as in the UI to tag variables and determine whether sensitive data is disclosed at sink points. Therefore, constant strings in the code and constant Ids that are associated with UI are of special interest and explicitly modeled in the language. For simplicity, we do not allow constant strings/Ids to appear in complex operations, *e.g.,* binary operations and method calls. For such scenarios, the constant is first assigned to a variable, which is further used in the complex operation. This is similar to how Android apps handle constant values in DEX bytecode. For example, the method call `json.getString("username")` at line 28 in Fig. 1 is converted to two statements: `tmp = "username"; json.getString(tmp);`.

An invocation to method `m`(v_f) is modeled by two separate statements: `call`(`m`, $v_a \rightarrow v_f$) passing the actual argument v_a to the formal argument v_f and `v=return`(`m`, v_r) returning the value in v_r in `m()` to `v` in the caller. The separation allows us clearly model the data flow at the entry and the exit of a method call. `v := apicall`(`m`, v_a) abstracts invocation to an API function `m()` whose implementation is usually excluded or not available during analysis, *e.g.,* the runtime C library and the framework methods for Android apps.

The language also supports conditional branches and loops. There are different loop structures such as `for` loops and `while` loops. We ignore these differences and use a `LOOP` statement to model them. Loop conditions are not relevant to our analysis and hence not modeled. Any side effects (in the loop conditions) are explicitly modeled as assignments in the loop body.

Our language is a kind of SSA language so that ϕ function is used to merge values from different branches (of a predicate). As we will show later in Section 3.2.2, ϕ functions require delicate consideration during bi-directional propagation.

3.2 Type System and Bi-directional Propagation

As discussed earlier, we use the constant texts in either the code

$$\text{Const-Binding}\ \frac{}{\Gamma, v := t \models \Gamma \Rightarrow [v : \{t\}]\Gamma}$$

$$\text{UI-Binding}\ \frac{resource_id(i)}{\Gamma, v := i \models \Gamma \Rightarrow [v : extract_text(i)]\Gamma}$$

$$\text{Unary-Assignment}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v_1 : T'}{\Gamma, v := \ominus v_1 \models \Gamma \Rightarrow [v : T \cup T', v_1 : T' \cup T]\Gamma}$$

$$\text{Binary-Assignment}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v_1 : T_1 \quad \Gamma \vdash v_2 : T_2}{\Gamma, v := v_1 \oplus v_2 \models \Gamma \Rightarrow [v : T \cup T_1 \cup T_2, v_1 : T_1 \cup (T - T_2), v_2 : T_2 \cup (T - T_1)]\Gamma}$$

$$\text{Phi-Assignment}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v_1 : T_1 \quad \Gamma \vdash v_2 : T_2}{\Gamma, v := \phi(v_1, v_2) \models \Gamma \Rightarrow [v : T \cup T_1 \cup T_2, v_1 : T_1 \cup (T - T_2), v_2 : T_2 \cup (T - T_1)]\Gamma}$$

$$\text{Method-Call-Param}\ \frac{\Gamma \vdash v_a : T \quad \Gamma \vdash v_f : T'}{\Gamma, call(m, v_a \rightarrow v_f) \models \Gamma \Rightarrow [v_f : T' \cup T, v_a : T \cup T']\Gamma}$$

$$\text{Method-Call-Return}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v_r : T'}{\Gamma, v := return(m, v_r) \models \Gamma \Rightarrow [v : T \cup T', v_r : T' \cup T]\Gamma}$$

$$\text{API-Call}\ \frac{\Gamma \vdash v_a : T' \quad \Gamma \vdash v : T}{\Gamma, v := apicall(m, v_a) \models \Gamma \Rightarrow [v : T \cup model_fwd(m, v_a), v_a : T' \cup model_bwd(m, v)]\Gamma}$$

Figure 4: Bi-directional propagation rule.

or the UI to tag the correlated variables and propagate the tags bi-directionally. We formalize this approach in a type system, *i.e.,the set of tags associated with a variable is treated as the type of the variable*. Since the type is a set, we also call it a *type set* in this paper. The mappings from variables to their type sets form the context Γ of the type system, which is iteratively updated during analysis until a fixed point is reached. For example, at the beginning, Γ is empty. Upon a statement `tmp = "username"`, Γ is updated to {*tmp* : {*username*}}. At this point, we have $\Gamma \vdash$ *tmp* : {*username*}, which means under context Γ, variable `tmp` is typed with set {*username*}. In other words, Γ(*tmp*) = {*username*}, where Γ(*tmp*) evaluates variable `tmp` in the context to obtain the corresponding type set.

When a statement is evaluated, the context may be updated. We use $\Gamma, S \models \Gamma \Rightarrow \Gamma'$ to indicate that under context Γ, evaluating statement S updates the context from Γ to Γ'.

We use [*var* : T]Γ to represent an update to the context. Specifically, if no mapping is found for variable `var` in context Γ, the mapping is added into the context. But if there exists some mapping for `var`, the rule substitutes the existing type set for `var` with the given type set T. Multiple mappings can be updated simultaneously, *e.g.,* [*var* : T, *var'* : T']Γ updates the context for two variables `var` and `var'`.

Given two type sets T and T', $T \cup T'$ unions the two sets while $T - T'$ returns a new type set which contains all elements belonging to T but not T'.

With the language in Fig. 3 and the above definitions, we define the bi-directional type set propagation rules in Fig. 4. The propagation is iterative. That means once the analysis starts, it does not terminate until the context Γ reaches a fixed point.

3.2.1 Binding Constant Value

As mentioned earlier, we focus on constant texts in the code and the constant Ids that are associated to UI. An assignment of a constant string to a variable adds a new mapping from the variable to a set holding the string to the context. For a constant Id, we need to make sure the Id is indeed a resource Id (*e.g.,* layout Id in Android apps or an Id for a specific UI element). This check is modeled by predicate `resource_id()`. If the prerequisite satisfies, updating the context is similar to the constant string assignment, except that the type set is the extracted texts from the corresponding UI through function `extract_text()`. For instance, if the constant Id is associated with a typical login screen, the extracted text set may often be {*Username*, *Password*, *Login*}.

3.2.2 Propagation for Assignment

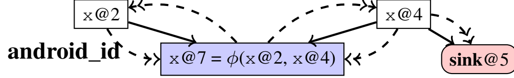
Rule Unary-Assignment updates the context for both the LHS and RHS variables with the union of the two separate type sets. Note that it allows the tags from LHS to propagate to RHS and vice versa through the union operation (*i.e.,* bi-directional propagation). Use the statement `jsonString = data` at line 18 in Fig. 1 as an example. Assume before evaluating this statement, Γ(*jsonString*) = {*avatar_url*, *username*, *created_at*, *content*} and Γ(*data*) = {*data*} via previous evaluation steps. After evaluating this statement, the type sets for both variables `jsonString` and `data` are updated to {*avatar_url*, *username*, *created_at*, *content*, *data*}. This shares some similarity with type unification in classic type inference. However, as we will see next, unification does not properly model the intended propagation behavior for binary operations and ϕ functions.

For a binary assignment, we cannot simply union all the type sets of the LHS and RHS variables and associate the resultant type set to all the variables, which is what classic type inference would do. We observe that this is undesirable as it allows the type set of a RHS variable to be propagated to another RHS variable while the operation does not induce any data flow between the two variables. Intuitively, assuming the two RHS variables are v_1 and v_2, v_1 being associated with a sensitive tag does not entail v_2 having the same sensitive tag (by the operation). Thus, as specified by Rule Binary-Assignment, the propagation is conducted as follows. The type sets of the RHS variables are unioned and inserted to the type set of the LHS variable. Only the part of the LHS type set that is not in the type set of v_1 is propagated to v_2 and only the part of the LHS type set that is not in the type set of v_2 is propagated to v_1. There is a corner case in which the two RHS variables are the same one, *e.g.,*`a = b ⊕ b`. The updated type set for `b` is $\Gamma(b) \cup (\Gamma(a) - \Gamma(b))$, which is equal to $\Gamma(a) \cup \Gamma(b)$. In other words, this special case behaves the same as a unary assignment. The propagation for ϕ statements has the same nature (Rule Phi-Assignment).

We use a real example from an Android app *com.mojo.animewallpaper* to show how our propagation rule for ϕ statements eliminates false alarms. The simplified code snippet is shown in Fig. 5a. If a certain condition satisfies, the device Id is assigned to variable `x` at line 2. The detail of acquiring the device Id is omitted but even-

```
1 if (...) {
2   x = getDeviceId(); // x is tagged with "android_id"
3 } else {
4   x = some_random_uuid(); // gen random value for x
5   Log.d("Random: ", x); // sink
6 }
7 use(x);
```

(a) Simplified code snippet.



(b) Data flow and type propagation.

Figure 5: Code example and bi-directional propagation for ϕ from app *com.mojo.animewallpaper*.

tually a constant string "*android_id*" is added to the type set of x. If the condition doesn't satisfy, a random value is generated as the requested Id at line 4 and stored to variable x, which is immediately used at a sink point at line 5. After the branch, variable x, whose value is either the real device Id or a random value, is used elsewhere.

From the perspective of ϕ representation, we know that right before the x is used at line 7, we have a ϕ statement as x@7 = ϕ(x@2, x@4). The data flow for the several occurrences of x is described by the solid arrows in Fig. 5b and the propagation relations are shown by dashed arrows.

Consider a naive bi-directional propagation that simply unions all the type sets. During the first iteration, "*android_id*" is propagated to x@7 via forward propagation. Nothing is backwardly propagated to x@2 or x@4 from x@7. Therefore, at the end of the first iteration, $\Gamma(x@2) = \Gamma(x@7) = \{android_id\}$ and $\Gamma(x@4) = \emptyset$. Then during the second iteration, if we directly propagate the type set of x@7 to both x@2 and x@4, we would get $\Gamma(x@4) = \{android_id\}$, which is later propagated to the sink point at line 5. Thus a sensitive data disclosure is reported which is a false alarm. In contrast, our propagation rule supports the mutual exclusion of the type sets in the two respective branches. Specifically, we only backwardly propagate $\Gamma(x@7) - \Gamma(x@2)$, *i.e.,* an empty set, to x@4. At last, the type set of x@4 stays unchanged and the sink point does not observe any sensitive type for the variable. Thus no sensitive data disclosure is reported.

3.2.3 Propagation for Method Calls

Propagation through a method call occurs at passing argument from the caller and returning value from the callee. Therefore, we define two separate rules for these two events. Note that these two rules handle method calls whose implementations are included in the analysis. We also propose a special rule for propagation over API functions, the implementations of which are typically invisible during analysis.

Rules Method-Call-Param and Method-Call-Return union the type sets. A concrete example for rule Method-Call-Param is the instantiation call at line 10 in Fig. 1. The constructor at line 17 is invoked and the value held by variable dt is passed to variable data. Then constant value "*data*" associated with dt is propagated to data and "*username*" associated with data is backwardly propagated to dt.

Rule API-Call does not directly propagate the type sets between parameters and the return value. BIDTEXT relies on the model for the API function for proper propagation. Prior static taint analysis [7, 14] have shown that it is effective to simply propagate from all parameters to the return value and the receiver object (*i.e.,*this reference in instance method calls). However, this naive approach does not work well in bi-directional propagation. We need to investigate the type correlations for the variables involved in an API call, including all the parameters and the return value.

Some API functions may not support fully bi-directional propagation among the variables. For example, variable name can be used to type value in statement value=HashMap.get(name) but not the reverse according to the semantics. Specifically, if name holds some sensitive constant strings, we can infer that value may hold sensitive information, but not the other way around. If we ignore this reference, after evaluating the statement under context Γ, we have $\Gamma'(name) = \Gamma(name)$ and $\Gamma'(value) = \Gamma(value) \cup \Gamma(name)$. Many API functions, on the other hand, can be applied with the naive propagation policy, unioning the type sets of all variables. For example, we have $\Gamma'(ret) = \Gamma'(str) = \Gamma(ret) \cup \Gamma(str)$ after evaluating statement ret = str.toUpperCase() under context Γ. In the rule, the behavior depends on functions model_fwd() and model_bwd() which define the propagation policies from v_a to v and from v to v_a, respectively.

We formalized our approach to identifying and bi-directionally propagating constant texts in a type system and developed a set of propagation rules based on our abstract language in Fig. 3. While the rules are general for our language, in practice we need to perform a number of enhancements to the rules to handle real-world language/program features. These enhancements are discussed in next section.

CheckAlert ::= IF(v_c) {alert(v_m)}

(a) Specialized statement.

$$\text{Check-Alert}\ \frac{\Gamma \vdash v_c : T \qquad \Gamma \vdash v_m : T'}{\Gamma, IF(v_c)\{alert(v_m)\} \models \Gamma \Rightarrow [v_c : T \cup T']\Gamma}$$

(b) Propagation rule.

Figure 6: Abstraction and propagation rule for Check-and-Alert cases.

3.3 Practical Enhancements

There are two main practical enhancements to our formal model that are critical to the effectiveness of BIDTEXT.

3.3.1 Check and Alert

It is common in real programs to prompt some alerts to the user or write to the log file if a condition check fails. In this case, we can use the alert/log message to infer what the corresponding variables involved in the condition check may hold. For example, an Android app can alert the user about some previous errors, *e.g.,* some required inputs are missing, by showing a short message on the screen. A typical implementation looks like the following.

```
1 if (str == null || str.isEmpty())
2   Toast.makeText(this, "Please Enter Password", 1);
```

We can type variable str with the constant text "*Please Enter Password*" and propagate it through the aforementioned rules.

The abstraction and the corresponding propagation rule are shown in Fig. 6. This applies to a set of API functions, called the *alert* functions.

3.3.2 String Concatenation

String concatenation is common in real-world apps. A concatenation operation may involve both constant values and multiple variables. If we simply union the type sets of all the involved variables and update the variables with resultant type set, we may introduce false positives. Furthermore, the associations between the constant strings (involved in the concatenation) and the variables (involved in the concatenation) also need to be properly identified.

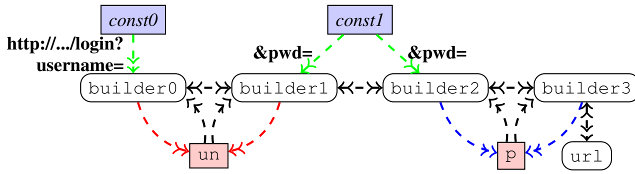


Figure 7: Propagation graph for a simple string concatenation.

$$\text{Str-API}\;\frac{api_w_str(m) \quad \Sigma \vdash v_a : E \qquad \Gamma' = string_partition(m, E)}{\Gamma, v := apicall(m, v_a) \models \Gamma \Rightarrow \Gamma \cup \Gamma'}$$

Figure 8: Propagation rule for string concatenation.

A simple strategy that associates all constant strings to all variables also produces a lot of false positives. For example, building a URL often involves multiple variables, each holding a value as part of the HTTP request. The variables can be either sensitive (*e.g.,* password) or insensitive (*e.g.,* user comment). We need to distinguish the exact types correlated to the variables. Consider the following example, in which a typical URL is constructed.

```
1 url = "http://.../login?username=" + un + "&pwd=" + p;
```

At the bytecode level, the above statement is converted to:

```
1 builder0 = new
      StrinBuilder("http://.../login?username=");
2 builder1 = builder0.append(un);
3 builder2 = builder1.append("&pwd=");
4 builder3 = builder2.append(p);
5 url = builder3.toString();
```

Assume the model for API `StringBuilder.append()` entails fully bi-directional propagation, *i.e.,* we propagate the type sets of all involved variables to each other. The constant string "*http://.../login?username=*" is propagated to `builder0`, `builder1`, `un`, `builder2` and `p`. A later text analysis would indicate that both `un` and `p` are associated with the sensitive text "*username*", which is incorrect for variable `p`. Similarly, "*&pwd=*" will be propagated to `un`, which causes a false alarm.

However, if we do not allow the propagation from the `StringBuilder` instance (*e.g.,* `builder0`) to the appended variable (*e.g.,*`un`), that is, the red and blue edges are removed from Fig.7, then neither "*username*" nor "*pwd*" could be propagated to `un` or `p`. As a result, we cannot infer that these two variables may hold sensitive information.

The expected propagation, according to the semantics of the URL string, is that "*username*" is propagated to `un`, and "pwd" to `p`, exclusively. We observe that it is impossible to enforce such propagation through API models (*e.g.,* the model for `append()`) as an API call may only represent a local operation that does not have the global view of the concatenated string. To address the problem, we need to analyze the entire concatenated string produced at the end. In our example, we ought to examine the final result associated with `url` in order to associate the appropriate text to variables `un` and `p`. Therefore, we need to enhance our type system with the following string analysis.

Rule Str-API in Fig. 8 determines if an API call has a string argument v_a with a well-defined format through function `api_w_str()`. For example, `new URL(str)` is such a function as it implies the variable `str` is a string of the url format. If so, the string is of interest. BIDTEXT computes an *abstract string* E for v_a, which is stored in a *string context* Σ that maps a variable to an abstract string. An abstract string is *a regular expression including both constant strings and variables*. The abstract string is partitioned by the function `string_partition()` so that the variables in the regular expression are associated with the appropriate

$$\text{Strcat}\;\frac{\Sigma \vdash v_1 : E_1 \qquad \Sigma \vdash v_2 : E_2}{\Sigma, v := strcat(v_1, v_2) \models \Sigma \Rightarrow [v : E_1 \cdot E_2]\Sigma}$$

$$\text{Strcat-Nil}\;\frac{\Sigma \vdash v_1 : nil \qquad \Sigma \vdash v_2 : E_2}{\Sigma, v := strcat(v_1, v_2) \models \Sigma \Rightarrow [v : v_1 \cdot E_2]\Sigma}$$

$$\text{Str-Const-Assign}\;\frac{}{\Sigma, v := t \models \Sigma \Rightarrow [v : t]\Sigma}$$

$$\text{Str-If}\;\frac{\Sigma, s_t \models \Sigma \Rightarrow \Sigma_t \qquad \Sigma, s_f \models \Sigma \Rightarrow \Sigma_f \qquad \Sigma_t \vdash v : E_t \qquad \Sigma_f \vdash v : E_f}{\Sigma, IF(*)\{s_t\}ELSE\{s_f\} \models \Sigma \Rightarrow [v : E_t \mid E_f]\Sigma}$$

$$\text{Str-LOOP-Closure}\;\frac{\bot, s \models \bot \Rightarrow \Sigma' \qquad \Sigma' \vdash v : v \cdot E \qquad \Sigma \vdash v : E_0}{\Sigma, LOOP\{s\} \models \Sigma \Rightarrow [v : E_0 \cdot (E)^*]\Sigma}$$

$$\text{Str-LOOP-Simple}\;\frac{\bot, s \models \bot \Rightarrow \Sigma' \qquad \Sigma' \vdash v : E \qquad v \notin E}{\Sigma, LOOP\{s\} \models \Sigma \Rightarrow [v : E]\Sigma}$$

Figure 9: Computing abstract strings.

texts. For the above example, the rule produces Γ' = {*un* : {*username*}, *p* : {*pwd*}}. We then combine Γ' into the current context Γ and further propagate the generated texts. Next, we will first explain how the abstract strings are computed and then the `string_partition()` function.

The rules for computing abstract strings are shown in Fig. 9. The interpretation of the rules is similar to that for our type system. One difference is that we use the string context Σ instead of the type context Γ. Rule Strcat simply concatenates the two abstract strings of the operands. Rule Strcat-Nil handles the case in which the first operand does not have any mapping, meaning that it is a string variable encountered for the first time. In this case, the variable itself is concatenated to the resulting string. It is similarly handled when the second operand does not have mapping and the rule is elided. Rule Str-Const-Assign handles the constant string assignment.

Rule Str-If specifies that for a conditional statement, BIDTEXT computes the string contexts for the true and false branches separately. For any variable that is present in the string context(s), the resulting abstract string is an alternation of the abstract strings in the branches. Consider the following code snippet.

```
1 if(c) str := strcat("&UserId=", uId);
2 else str := strcat("&sessionId=", sId);
```

The abstract string for variable `str` is ("*&UserId=*"·`uId`) | ("*&sessionId=*"·`sId`).

Rule Str-LOOP-Closure specifies that for a loop, BIDTEXT first computes the string context for the loop body with an empty string context and then aggregates the resulting abstract strings to the original string context. In particular, if the abstract string for a variable v also contains v, it indicates the resulting string has recursive structure (caused by the loop), BIDTEXT hence associates v to a kleene closure in the context outside the loop. Tail recursion is similarly handled. Currently, BIDTEXT only handles regular languages, which is sufficient for most cases we encountered. Rule Str-LOOP-Simple specifies that if there is no recursive structure, the abstract strings are simply copied from the context of the loop body to the context outside the loop. For the following example, BIDTEXT produces the abstract string "*Output:*"·("*A*")* for variable `str`.

```
1 str := "Output:";
2 for (...)
3   str := strcat(str, "A");
```

As shown by Rule Str-API, the abstract string at an API that specifies the format of the string is partitioned to acquire the texts for the variables within the abstract string. This is done by calling

`string_partition()`. This function has a number of built-in parsers that can parse the different string formats based on the API name. For example, if the API is `URL()`, it uses the parser for url. Particularly, the parser searches for symbol "*?*", the part after the symbol is parsed by "*([^=]*)=([^&]*)*" with the first part being the key and the second part the value. If the key is a constant t and the value is a variable v, Γ is updated with the mapping from v to t. BIDTEXT also has parsers for other formats such as SQL queries. For example, two mappings $\{$*v1* : $\{$*password*$\}$, *v2* : $\{$*userid*$\}\}$ can be extracted from an abstract string denoting a SQL update "*update TABLE set password=*"· v1· " *where userid=*"· v2.

For the prior URL example, `append()` is essentially a `strcat()`. According to the rules, the final abstract string for `url` is "*http://.../login?username=*"· un· "*&pwd=*"· p. It is partitioned so that `un` is mapped to $\{$*username*$\}$ and `p` is mapped to $\{$*pwd*$\}$.

3.4 Disclosure Analysis

After the type set computation converges, BIDTEXT checks whether arguments at the sinks points hold any sensitive data via textual analysis. If the type set information indicates the sensitiveness of an argument, we report a potential disclosure.

Algorithm 1 Sensitiveness determination.

determine_sensitiveness(T, S, *KWD*)
1: **for all** $t \in T$ **do**
2: t' = preprocess(t)
3: **if** t' matches in *KWD* **then**
4: **if** t' is a word **or** t' doesn't match any negation template **then**
5: $S = S \cup t$
6: **end if**
7: **end if**
8: **end for**

The process to determine the sensitiveness of a variable with a set of associated constant texts is presented in Algorithm 1, which assumes the text set T and a set of sensitive keywords *KWD*. For each collected string (*i.e.,* word, phrase or sentence), BIDTEXT first conducts some preprocessing. For example, "*EmailAddress*" is converted to "*email address*". If a string contains more than one sentence, it is split using the standard sentence division method implemented in Stanford Parser [34]. If the string matches any keyword, we check whether it is a single word. If so, we put the string into S which holds all sensitive strings. S can be used to decide what sensitive information is disclosed after the algorithm finishes. If the string is a phrase or a sentence, we need to check if it is the negation of a sensitive keyword. For example, "*do not enter password here*" tells the user that the input field should not contain any password. Even though the string matches a sensitive keyword "*password*", we do not consider it sensitive. So if the corresponding variable does not have any other associated sensitive texts, it is treated insensitive and the sink does not have a sensitive data disclosure problem.

We use Stanford Parser [34] to parse a phrase or a sentence into a syntax tree, which is then converted to a dependency relation (please refer to [5]). Based on the dependency relation, BIDTEXT searches the negation word "*not*" and then checks the auxiliary word right before the negation word. It also examines if there exists a subject noun word before the auxiliary word. By combining the auxiliary word and the possible subject word, BIDTEXT can identify whether the phrase/sentence is imperative or declarative. For example, "*do not*" and "*you should not*" are imperative negations but "*you did not*" is declarative negation. BIDTEXT only considers the imperative negation as a negation (of sensitive keyword). In such cases, the text is not sensitive.

4. IMPLEMENTATION

We implemented BIDTEXT to detect sensitive data disclosures in Android apps. BIDTEXT is built on top of WALA [6], which parses the Android DEX bytecode to intermediate representations. We implemented the algorithm in [23] to collect possible entry points (*e.g.,*`onCreate` for an *activity*) in the target Android app. For each entry point, BIDTEXT builds the call graph and the dependency graph. The constant strings are propagated on the graphs. We do not distinguish the correlated text for each UI element as in [17]. Instead, all elements in one layout file are associated with all the texts found in that layout file.

BIDTEXT relies on a keyword set to determine the sensitiveness of computed texts. To acquire the keyword set, we ran BIDTEXT on 2,000 randomly selected apps and extracted all texts discovered for each sink. We then manually inspected these texts to construct the keyword set. In order to detect traditional data closures that are due to data-flow between source APIs and sink APIs instead of texts, we assign some sensitive textual keywords to the source APIs that must expose sensitive information so that BIDTEXT can propagate the keywords. For example, we assign "*imei*" to API `TelephonyManager.getDeviceId()`.

We leverage Stanford Parser [34] as the engine for analyzing phrases and sentences. BIDTEXT currently only supports English.

For better efficiency, BIDTEXT also performs on the fly type set reduction. Specifically, when a text set reaches a certain size, garbage collection is conducted by filtering out the texts in the type set that do not indicate sensitiveness and those that are redundant.

5. EVALUATION

All experiments are performed on an Intel Core i7 3.4GHz machine with Ubuntu 12.04. The task of analyzing each app is given the maximum memory of 10GB and the maximum analysis time of 20 minutes. The subjects are a collection of 10,000 Android apps downloaded from Google Play in March 2015. The sink points used in the evaluation contain all the logging operations in Android and the Apache HTTP access APIs that are commonly used in Android apps. This is also the standard setup for many existing static taint analysis [17, 18]. The other types of sink points can be easily added to BIDTEXT.

5.1 Pilot Study

As discussed earlier, BIDTEXT heavily relies on accurate propagation models for API method calls. However, Android framework contains thousands of API functions, making it almost infeasible to manually build the models for all API functions. Our approach is to randomly select 2,000 apps and run BIDTEXT on these apps. Then we inspect the results to discover popular API functions and create models only for those functions. These models are later used in the larger scale study.

During the pilot study, we also observe a kind of false positive that appears frequently. It is caused by a Facebook library used by many apps. The library logs an error message when it fails to obtain the device Id. The code snippet is abstracted as follows.

```
1 try { /* acquire device id */ }
2 catch (Exception e) { Utility.logd("android_id", e); }
```

The message `e` is typed with "*android_id*", which is a sensitive keyword. But the meaning of this message is indeed that the action of acquiring the device Id fails. Solving this issue requires in-depth semantic analysis of `e` which is not supported by BIDTEXT. Since the pattern is fixed, we post-process all the reports to filter out this pattern for both the pilot study and the later large scale study.

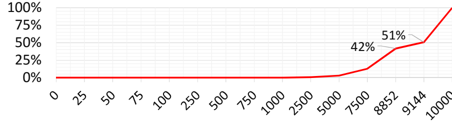


Figure 10: Distribution of accumulative analysis time for all apps.

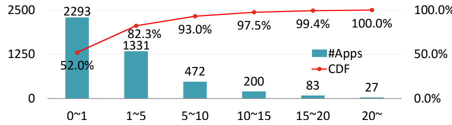


Figure 11: Distribution for the analysis time (in minutes) of the apps reported with sensitive data disclosures.

5.2 Unification vs. Bi-directional Propagation

In classic type inference, given an assignment statement such as `z=x+y` and `z=`ϕ`(x,y)`, the updated type sets of `x`, `y`, and `z` are the union of all three original type sets. In Section 3.2.2 (Rules Binary-Assignment and Phi-Assignment in Fig. 4), we mentioned that such a unification based approach may produce a lot of false positives and hence BIDTEXT makes use of a bi-directional propagation strategy that avoids propagating type sets between right-hand-side operands (*i.e.,*`x` and `y` in the example). In this experiment, we want to compare these two propagation strategies.

Due to the lack of ground truth, such a study requires manually inspecting the reported disclosure defects and determining if they are false positives. Among the 2000 apps tested in the pilot study, we selected the first 60 apps whose *data disclosure path* (*i.e.,* the data flow subgraph that includes the path from the source to the sink and the path that the sensitive text is propagated from its origin to the sink) involves ϕ statements and/or binary operations with the unification based propagation policy. We re-run BIDTEXT on the 60 apps with the bi-directional propagation policy and compare the two sets of results.

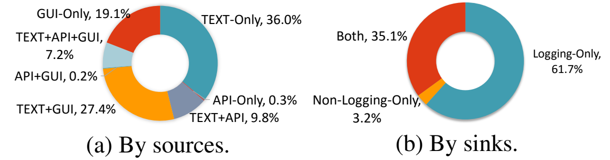
Among these 60 apps, 42 of them are reported by both the unification policy and the bi-directional policy; 25 of them contains flows only reported by the unification policy. Note that the two do not add up to 60 because some apps have multiple reported disclosures, some being reported by both policies and the others being only reported by the unification policy. We manually studied the 25 cases reported by the unification policy and found that they are all false positives. We have shown one sample false positive in Section 3.2.2.

5.3 Large Scale Evaluation

In this experiment, we use 10,000 apps not covered by the pilot study. The apps have a minimum size of 6.46KB for the APK files and a maximum size of 49.94MB. The average size of the APK files is 9.17MB. Among these apps, there are two that do not contain any DEX bytecode in the APK files. For the remaining apps, the minimum size of the bytecode files (`classes.dex`) is 452 bytes and the maximum size is 10.32MB. The average size of the bytecode files is 2.53MB.

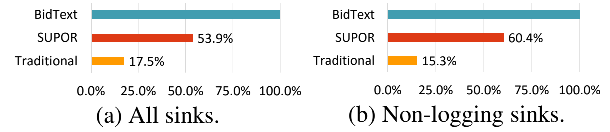
5.3.1 Results

The total analysis time for the 10,000 apps is 587.6 hours. Fig. 10 presents the distribution of the cumulative analysis time for all the 10,000 apps. We divide the total analysis time into three parts according to how the analysis on an app terminates. As mentioned above, we set the analysis timeout to 20 minutes for each app. In our evaluation, 856 apps (8.56%) time out and the total analysis time account for 49% of the total time consumed for the 10,000



(a) By sources. (b) By sinks.

Figure 12: Breakdown of the reported apps.



(a) All sinks. (b) Non-logging sinks.

Figure 13: Comparing BIDTEXT with static tainting (tracking specific APIs) and SUPOR [17].

apps. We have 293 other apps of which the analysis ran out of memory. The total time for these apps accounts for 9%. For the remaining 8,852 apps that finished normally take only 42% of the total analysis time. Observe in Fig. 10 that the first 7,500 apps take less than 15% of the total time. Among the 8,852 apps, the minimum analysis time is 0.2 seconds and the maximum time is 1197.4 seconds. The median is 24.9 seconds while the average time is 99.9 seconds. The largest app that terminates normally has the APK size of 49.94MB, and the bytecode size of 10.32MB.

Overall, BIDTEXT reports 4,406 apps with sensitive data disclosure problems. We show the analysis time distribution of these apps in Fig. 11. The blue bars show the number of apps that finished within a time period. For instance, 472 apps took more than 5 minutes but less than 10 minutes. We also see that 27 apps timed out in the experiments, although partial results were collected before the analysis terminated. The red line presents the cumulative analysis time: 93.0% of the apps were analyzed within 10 minutes. We can conclude that BIDTEXT is efficient to be applied to market-scaled apps.

We also show the breakdown of the 4,406 apps by the sources of data disclosures in Fig. 12a.

There are three types of sources: (1) TEXT – constant texts in the code that denote sensitive data; (2) API – sensitive API (recall that BIDTEXT also detects data disclosures originating from sensitive APIs by associated artificial texts to the source APIs such as `Location.getLatitude()`); and (3) UI – constant texts retrieved from user interfaces that denote sensitive data. Observe that the majority of disclosures are/can be detected by the sensitive text labels. Some data disclosure defects can be recognized through multiple sources (*e.g.,* TEXT+API), meaning that there are some (bi-directional) data flow paths from a sensitive API to a sink and from some constant text to the same sink. Consider the following example. The data flow path 2→6→7 denotes a disclosure originating from TEXT (*i.e.,* "*android_id*") and the path 4→6→7 denotes a disclosure originating from API (*i.e.,* "`getDeviceId()`").

```
1 if (fails_to_obtain_imei()) {
2   id = Settings.Secure.getString(resolver,
          "android_id");
3 } else {
4   id = telephonyManager.getDeviceId();
5 }
6 json.putString("id", id);
7 http_sink(json.toString()); // sink
```

The breakdown of the apps by the sink types is shown in Fig. 12b. Note that 64.9% of the reported apps contain disclosures due to logging. Although data disclosure through logging is substantially

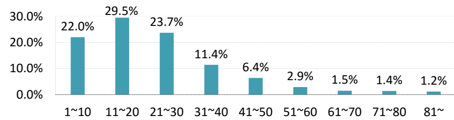


Figure 14: Length distribution of the emitted paths for the reported apps. X axis shows the length of the paths.

Table 1: Manually inspected evaluation results for 100 apps.

	TEXT	API	UI
Total	84	22	39
Only	44	2	14
FP	3	0	7

mitigated by access control in the latest version of Android, it is still a security concern for legacy Android systems such that most existing works [7, 22, 17] report these disclosures. About 38.3% of the reported apps (16.9% of all the apps evaluated) contain sensitive data disclosures due to to non-logging sinks. They are serious threats even in the latest Android systems.

Fig. 13 shows how BIDTEXT compares with an implementation of the traditional taint tracking technique (tracking disclosures from source APIs through forward data-flow similar to [14]) and SUPOR [17], which is a technique that tracks disclosures from sensitive UI elements (*e.g.,* input boxes) through forward data-flow. BIDTEXT always reports a super-set of those reported by the classic tainting and SUPOR. In the figure, the numbers of apps reported by tainting and SUPOR are normalized to those reported by BIDTEXT. Observe that they only report 17.5% and 53.9% of those reported by BIDTEXT, respectively. Even combining the two can only detect 64.0%. If only taking non-logging disclosures into account, they report 15.3% and 60.4% of those reported by BIDTEXT. This attributes to both the new text label correlation analysis and the bi-directional type set propagation strategy.

We present the length distribution of the emitted data disclosure paths for the 4,406 apps in Fig. 14. Though some paths tend to be very long (more than 80 elements), most of them are relatively short. More than 75% of the paths require less than 30 steps from the origination of the sensitive texts to the sink points.

False Positives and False Negatives. It is critical to understand the quality of the reported defects. Due to the lack of ground truth, we had to perform manual inspection. Studying the full set of results is infeasible. Hence, we randomly chose 100 reported apps with a uniform size distribution for manual inspection. The results are presented in Table 1.

The columns indicate the sources of the disclosures. Row *Total* shows the total number of reported apps for each sources. Row *Only* shows the number of apps that only have reported disclosures falling into one category. The last row shows the number of false positives.

Observe that the 10 false positives are exclusive. Therefore, the false positive rate is 10%. The causes for false positives will be discussed in Section 5.3.3. We do not count the false negatives because we don't have the ground truth.

Among the 84 apps where disclosures are reported by code text analysis, 62 apps contain paths that can be only detected by our approach via text correlation analysis, *i.e.,* the data used at sink points neither come from any UI inputs nor from traditional source APIs. In other words, 62 of them cannot be detected by classic tainting or SUPOR. This ratio is consistent with that in Fig. 13 for the larger experiment. The other reported disclosures have the sensitive data coming from these two categories of sources. They are reported by both BIDTEXT and the existing technique(s). Another interesting finding is that BIDTEXT often produces a shorter disclosure path. A typical scenario is that there is a long data flow path from a UI input element to a sink. However, mid way through the path, the (sensitive) data is put/get to/from some container with a sensitive textual key, which allows BIDTEXT to report a shorter path from the put/get operation to the sink. The benefits of shorter paths are two-folded: less human efforts needed for inspection and detecting more disclosures (because the full path from the source points to the sink points might be complicated, involving inter-component communications, such that the tool may fail to traverse the full path).

5.3.2 Case Studies

We observe many cases in which sensitive textual keys appear together with data in key-value operations, *e.g.,* constructing a name value pair (*e.g.,com.gunsound.eddy.fafapro*), inserting data into a hash map (*e.g.,me.tango.fishepic*), retrieving/adding data to persistent storage through an instance of `SharedPreferences` (*e.g., com.ifreeindia.sms_mazaa*) or putting data into a JSON object (*e.g., com.mobilegustro.war.battle.air.force*). BIDTEXT recognizes the sensitiveness of corresponding data via text correlation analysis.

In the following, we show a code snippet adopted from app *com.-pro.find.differences* that discloses sensitive device information to Web servers.

```
1  void obtainDeviceInfo() {
2    TCore.aid = Settings.Secure.getString(resolver,
         "android_id");
3  }
4  void connectWebServer() {
5    Map map = new HashMap();
6    safePut(map, "android_id", TCore.aid);
7    String params = convertURLParams(map); // omitted
8    http_sink(params); // sink
9  }
10 void safePut(Map map, String k, String v) {
11   map.put(k, v);
12 }
```

The method call at line 2 returns system information based on the given key value. For example, a unique Id for the device is obtained if "*android_id*" is given as the key. If the key is "*enabled_input_methods*", the return value contains a list of input methods that are currently enabled. Therefore, the sensitiveness of the return value depends on the key. BIDTEXT works by correlating the textual key with the return variable to decide whether a later sink operation involves sensitive data or not.

In the above example, the variable `TCore.aid` is typed with the constant text "*android_id*" at line 2, which is later propagated to parameter `v` of method `safePut()` at line 10. `v` is inserted into the hash map at line 11. Note that "*android_id*" at line 6 is propagated to `k@10` which is further propagated to the hash map and variable `v` according to the corresponding API model for propagation. Along the data flow, the constant text is propagated to `params@7` that is eventually used at the sink point at line 8. BIDTEXT reports the data disclosure.

5.3.3 False Positives

One of the 10 false positives is caused by unmodeled API functions. The corresponding code snippet is from app *at.zuggabecka.-radiofm4*.

```
1  uidx = cursor.getColumnIndex("username");
2  iidx = cursor.getColumnIndex("_id");
3  id = cursor.getLong(iidx);
4  sink(id);
```

At line 1, a sensitive keyword "*username*" is correlated with the receiver object `cursor` that is related to a database query. Then all uses of `cursor` propagate the text label to other variables, *e.g.,* the return value of a relevant method call. Thus, `id` at line 3 is typed

with "*username*". Later when it is used at a sink point, BIDTEXT reports a sensitive data disclosure after analyzing the corresponding type set. To remove this false alarm, we can build a model for API `Cursor.getColumnIndex(key)` to only propagate type set from `key` to the return value, avoiding propagating to the receiver object. Then in the above code snippet, only variable `uidx@1` is typed with "*username*". Variable `id` that appears at the sink point is only typed with "*_id*" which is not considered as a sensitive keyword. Therefore there is no disclosure problem with the model.

All the other nine false positives are caused by incorrect recognition of text, two for code text and seven for UI text.

App *com.netcosports.andalpineski* contains a text label as "*Apps-_lang[apps_lng_iso2]*" which indicates the language of the app. However, it contains a predefined sensitive keyword "*lng*" which is mostly used as an abbreviation of "*longitude*". Failing to understand the meaning of the text, BIDTEXT incorrectly reports a sensitive data disclosure.

App *com.wactiveportsmouthcollege* has a UI text of "*Pin to desktop*" where sensitive keyword "*Pin*" is used as a verb. Failing to understand it leads to a false positive. All other false positives have similar causes – sensitive keywords in a phrase or sentence do not indicate any sensitive information. Possible solutions for this type of false positives include integrating more advanced NLP techniques with program analysis to understand the meanings of the text.

5.4 Discussion

One limitation of BIDTEXT lies in that the text in code may not be in a generalized format. For example, some developers use "*lng*" for "*longitude*" whereas others use "*long*" for it, which is a more general word in English. If we treat "*long*" as a sensitive keyword, we can expect many false positives. In addition, developers tend to combine several words (or abbreviations) into a single word, which makes it more difficult to determine whether the correlated data are sensitive or not.

In the future, we plan to improve our approach in the following aspects. The first one is to discover text labels in the names of method calls, if they are not obfuscated, and variable/field names. The second improvement is to consider code comments if source code is available. The third one is to improve the NLP aspect by putting the keywords in their program context. Doing so, we may be able to recognize "*long*" indeed means longitude.

6. RELATED WORK

A lot of prior research has focused on detecting sensitive data disclosures, either statically or dynamically, for mobile apps [14, 7, 8, 9, 15]. Most of them consider specific APIs as sensitive source points while BIDTEXT analyzes text labels to determine if a variable can hold sensitive data. SUSI [32] gives a comprehensive list of the data sources in Android, but it does not assume the data obtained from the sources must be sensitive. In addition, even if the state-of-the-art static detectors, *e.g.,* FlowDroid [7] and DroidSafe [15], had been enhanced with various ways of determining data sensitiveness, they would likely not be able to detect some sensitive data disclosures reported by BIDTEXT such as our motivating example, where the sensitiveness of the data is determined after the sink point and there is no forward data-flow from the sensitiveness revelation point and the sink point. BIDTEXT, however, leverages bi-directional propagation to address this problem.

Huang *et al.* developed type-based taint analysis to detect information leaks in Java-based Web applications and Android apps via type inference [19, 20]. They abstract the information flow analysis into a type system and check if any type error occurs. Their technique scales well without using advanced points-to analysis [7, 15]. Their technique still follows the traditional definition of data disclosure, which is a forward data flow path from the source to the sink. In other words, it does not propagate data sensitiveness in a backward fashion. As such, it may not be able to report many disclosures reported by BIDTEXT, including the motivating example. Furthermore, their type system does not leverage text information. Ernst *et al.* also developed a type-based taint analysis system [11]. Their technique associates a few (security) types such as LOCATION, INTERNET, and SMS to sources and sinks and have a set of predefined policies such as LOCATION can only be compatible, or type-checked, with INTERNET. So if LOCATION reaches a program point with the SMS type, a leak is reported. Their flow analysis is forward whereas BIDTEXT is bi-directional. And BIDTEXT leverages text labels.

SUPOR [17] and UIPicker [27] discover sensitive information on user interfaces through static analysis. However, they essentially belong to the traditional forward data-flow based techniques. AsDroid [18] collects the set of API calls in an event handler and compares the meaning of these API calls with the UI text of the event to detect unwanted/unexpected app behavior. In contrast, BIDTEXT types individual variables in the program with text labels and leverages a type system that allows bi-directional propagation. Researchers also combine code and comment analysis to detect bugs or inconsistencies [35, 36, 37]. We envision comment analysis can leverage our bi-directional type system so that the information in comments can be leveraged to analyze fine-grained and in-depth app behavior. In addition, WHYPER [29] and AutoCog [31] apply NLP techniques to app's descriptions to obtain a comprehensive view of the app and check if the required permissions are appropriately specified in the descriptions. Besides, [30] and [41] apply NLP techniques on API descriptions or documents to infer method specifications. We can leverage these techniques to automate the generation of API models used in BIDTEXT.

7. CONCLUSION

We propose BIDTEXT, a novel static technique to detect sensitive data disclosures. BIDTEXT identifies text labels appearing in both code and UI, treats them as types, associates them to the corresponding variables, bi-directionally propagates the types through data flow and eventually attributes them to sink points that potentially disclose sensitive information. At the end, the parameters at the sink points have type sets of correlated texts. Textual analysis is applied to the type sets to determine if the variables may hold sensitive data. We implement BIDTEXT and evaluate it on 10,000 apps downloaded from Google Play store. BIDTEXT reports 4,406 apps that have sensitive data disclosure problems including 4,263 apps disclosing sensitive information through logging and 1,688 through non-logging channels. Existing techniques can only report 64.0% of cases reported by BIDTEXT. Manual inspection shows the false positive rate is 10%. The overhead of BIDTEXT is reasonable.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, Cisco Systems under an unrestricted gift, and Natural Sciences and Engineering Research Council of Canada. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

9. ARTIFACT DESCRIPTION

This artifact provides an executable environment of BIDTEXT. The goal is to reproduce the results shown in the paper, as well as to let the users be able to apply BIDTEXT on Android apps of their interest.

9.1 Where is the artifact?

The source code is publicly available at https://bitbucket.org/hjjandy/toydroid.bidtext. Gradle [3] build is supported. And JDK 7 or newer versions is required to compile the source code.

A virtual machine (VM) image with Ubuntu 14.04 can be found at: https://github.com/hjjandy/FSE16-BidText-Artifacts-VM. It is created by Oracle VM VirtualBox [4]. The VM contains the source code, executables and a small set of test apps. It has been tested on Windows 10 and Ubuntu 14.04, but not on MacOs.

The users can also download the executables and the test apps at https://github.com/hjjandy/FSE16-BidText-Artifacts. It provides support to execute the tool on both Windows and Linux if Java 7 or newer versions with AMD64 architecture has been installed. Git LFS [2] is required to clone the repository. If the users encounter an `IOException` when BIDTEXT tries to load the parser from *englishPCFG.ser.gz*, please check if the models of Stanford Parser ("*stanford-parser-3.4.1-models.jar*") have been downloaded correctly via Git LFS.

9.2 What are contained?

In the HOME directory of the VM, one file and three folders are related to the artifacts.

File "*FSE16-Artifacts-Eval-README*" is a detailed description of the artifacts, including how to run the tool and how to understand the results. It also contains some issues that need special attention to run the tool well.

Folder "*BidText-Source*" contains a full copy of the source code. The users can update it via "*git pull*". "*BidText-TestApps*" contains a portion of Android apps used in the paper. Inside it, there are four special cases, corresponding to the four examples presented in the paper. They are the motivation example (Fig. 1 in Section 2), the example for bi-directional propagation on PHI statement (Fig. 5 in Section 3.2.2), the example of two different types of sources flowing to the same sink (Section 5.3.1) and the case study in Section 5.3.2. Besides, sub-folder "*Others*" contains the 100 apps used to measure the accuracy of BIDTEXT, whose results are presented in Table 1. "*BidText-Bin*" contains the necessary libraries, configurations and scripts to execute BIDTEXT. Note that it is infeasible to host all the apps used in the paper due to the sheer volume of the apps.

9.3 How to run the tool?

In order to apply the tool on an Android app, the user can run it on a terminal under the folder "*BidText-Bin*": `./RUN Path_to_APK`. The user can also use the four start scripts to perform analysis on the four corresponding cases. For example, `./Motivation` is equal to `./RUN $HOME/BidText-TestApps/Motivation/com.buycott.android-22.apk`. If the users want to test the 100 apps, just execute `./Eval-100`.

Since BIDTEXT requires a lot of memory to perform analysis, the user is required to allocate enough memory for the VM when creating a VM in VirtualBox. It should be at least 5GB because the default setting of JVM heap size for running BIDTEXT in the VM is 4GB, which is not enough to evaluate all the 100 apps. We suggest 12GB to evaluate the 100 apps. The users can modify "*bidtext.prop*" to set a larger JVM heap size for BIDTEXT.

The non-VM artifacts also contain start scripts (batch files) to allow easy execution of BIDTEXT on Windows. The commands and the settings are the same as in the VM.

9.4 How to understand the results?

After analyzing an APK file, BIDTEXT generates the results (if reported any) in folder "*APK_name*.bidtext" which is aside the APK file. Each reported sink point has an individual result file named "*idx.Sink_Type*.txt", *e.g.,* 384.LOG.txt. The result file contains the sink API and the enclosing method of the sink. For each identified sensitive textual label, surrounded by "********", the propagation path is listed. Each path element is the `String` representation of a WALA [6] `Statement`, an IR used during the analysis. In the following, we show part of the results for the motivation example. Note that we did some simplification for readability.

```
1 ********username********
2 NORMAL_RET_CALLER:Node: <CampaignActivity$20$1,
      run()V> 66 = invokevirtual <JSONObject,
      getString(String)String> 18,64
3 NORMAL_RET_CALLER:Node: <CampaignActivity$20$1,
      run()V> 18 = invokevirtual <JSONArray,
      getJSONObject(int)JSONObject> 5,172
```

NORMAL_RET_CALLER is the type of the `Statement` and "`Node`" indicates the location of the statement. The numbers are the variables values, distinct for any SSA variables. For example, "`66`" can be treated as a variable `v66`, which can only be defined once in the enclosing method. For more details about the representation, please refer to WALA document. The sensitive textual label "*username*" propagates from line 1, which is a method invocation. Through inspecting the app's code, we know that the label is a constant string stored in `v64` while the JSON object is associated with `v18`. In the second statement, we see that `v18` is the return value of a method call. That means, the two statements constitute a *backward* propagation. Please refer to Fig. 1 to get a better understanding of the motivation example.

10. REFERENCES

[1] Buycott. http://buycott.com/. Accessed: 03 Mar 2016.

[2] Git large file storage. https://git-lfs.github.com/.

[3] Gradle build tool | modern open source build automation. https://gradle.org/.

[4] Oracle VM VirtualBox. [https://www.virtualbox.org/.

[5] Universal dependencies. http://universaldependencies.github.io/docs/.

[6] WALA: T.J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[7] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI* (2014).

[8] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting privacy leaks in iOS applications. In *NDSS* (2011).

[9] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[10] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *USENIX Security* (2011).

[11] ERNST, M. D., JUST, R., MILLSTEIN, S., DIETL, W., PERNSTEINER, S., ROESNER, F., KOSCHER, K., BARROS,

P. B., Bhoraskar, R., Han, S., Vines, P., and Wu, E. X. Collaborative verification of information flow for a high-assurance app store. In *CCS* (2014).

[12] FortiGuard Center. Information disclosure vulnerability in OpenSSL (Heartbleed). http://www.fortiguard.com/advisory/2014-04-08-information-disclosure-vulnerability-in-openssl.

[13] Gartner. Gartner says smartphone sales surpassed one billion units in 2014. http://www.gartner.com/newsroom/id/2996817.

[14] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST* (2012).

[15] Gordon, M. I., Kim, D., Perkins, J., Gilhamy, L., Nguyenz, N., and Rinard, M. Information-flow analysis of Android applications in DroidSafe. In *NDSS* (2015).

[16] Haris, M., Haddadi, H., and Hui, P. Privacy leakage in mobile computing: Tools, methods, and characteristics. *CoRR abs/1410.4978* (2014).

[17] Huang, J., Li, Z., Xiao, X., Wu, Z., Lu, K., Zhang, X., and Jiang, G. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security* (2015).

[18] Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B. Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ICSE* (2014).

[19] Huang, W., Dong, Y., and Milanova, A. Type-based taint analysis for Java Web applications. In *FASE* (2014).

[20] Huang, W., Dong, Y., Milanova, A., and Dolby, J. Scalable and precise taint analysis for Android. In *ISSTA* (2015).

[21] Kim, J., Yoon, Y., and Yi, K. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *MoST* (2012).

[22] Lu, K., Li, Z., Kemerlis, V., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., and Jiang, G. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS* (2015).

[23] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS* (2012).

[24] Mahemoff, M. "Offline": What does it mean and why should I care? http://www.html5rocks.com/en/tutorials/offline/whats-offline/.

[25] MITRE. CWE-200: Information exposure. https://cwe.mitre.org/data/definitions/200.html.

[26] Mulliner, C. Privacy leaks in mobile phone internet access. In *ICIN* (2010).

[27] Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., and Wang, X. UIPicker: User-input privacy identification in mobile applications. In *USENIX Security* (2015).

[28] OWASP. Information leakage. https://www.owasp.org/index.php/Information_Leakage.

[29] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security* (2013).

[30] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., and Paradkar, A. Inferring method specifications from natural language API descriptions. In *ICSE* (2012).

[31] Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., and Chen, Z. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *CCS* (2014).

[32] Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)* (2014).

[33] Search Engine Watch. Mobile now exceeds PC: The biggest shift since the Internet began. https://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began.

[34] The Stanford Natural Language Processing Group, 1999. http://nlp.stanford.edu/.

[35] Tan, L., Yuan, D., Krishna, G., and Zhou, Y. /* iComment: Bugs or bad comments? */. In *SOSP* (2007).

[36] Tan, L., Zhou, Y., and Padioleau, Y. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *ICSE* (2011).

[37] Tan, S. H., Marinov, D., Tan, L., and Leavens, G. T. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *ICST* (2012).

[38] US-CERT. OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). https://www.us-cert.gov/ncas/alerts/TA14-098A.

[39] Xia, N., Song, H. H., Liao, Y., Iliofotou, M., Nucci, A., Zhang, Z.-L., and Kuzmanovic, A. Mosaic: Quantifying privacy leakage in mobile networks. In *SIGCOMM* (2013).

[40] Zakas, N. C. Towards more secure client-side data storage. https://www.nczonline.net/blog/2010/04/13/towards-more-secure-client-side-data-storage/.

[41] Zhai, J., Huang, J., Ma, S., Zhang, X., Tan, L., Zhao, J., and Qin, F. Automatic model generation from documentation for Java API functions. In *ICSE* (2016).