2012 19th Working Conference on Reverse Engineering

AQUA: Android QUery Analyzer

Chon Ju Kim
Computer Science and Engineering
Polytechnic Institute of NYU
6 Metrotech Center
Brooklyn, New York 11202
email: ckim01@students.poly.edu

Phyllis Frankl
Computer Science and Engineering
Polytechnic Institute of NYU
6 Metrotech Center
Brooklyn, New York 11202
email: pfrankl@poly.edu

***Abstract*—**

Smartphone and tablet users typically store a variety of sensitive information on their devices, including contact information, photos, SMS messages, and custom data used by various applications. On Android devices, the data is stored in SQLite databases which applications access by constructing and executing queries, either directly or via Android content provider API calls. Before installing an application that uses a content provider, a user must grant permission for the application to read and/or write the associated data. Many users grant permission with little understanding of the risks. Even more savvy users cannot make well-informed decisions, as they are only given very coarse information about what data the application accesses.

To provide users with more detailed information about how Android apps access and modify stored data, we have developed AQUA, the Android QUery Analyzer. AQUA analyzes application binary code, performing a lightweight static analysis to determine possible values of string variables that are incorporated into queries. AQUA reports on the content providers used and the database tables/attributes accessed and/or updated, allowing users to make more informed decisions about whether to grant permissions. This paper describes AQUA's design and evaluates AQUA's accuracy and performance by using it to analyze 105 popular apps downloaded from Google Play.

***Keywords*-Android; Database application; Static analysis;**

I. Introduction

While Android platforms and applications for them are exploding in popularity, there is increasing awareness that these applications can pose threats to security and privacy [1]. Many applications access private data that is stored in databases on the Android device, such as information about the users contacts, photos, sms messages, etc. Application developers must declare resources that the application uses in a *manifest* file and an Android application cannot be installed unless the user explicitly grants permission for the application to access those resources. This appears to protect users from rogue applications. Unfortunately, users have little information about what the applications actually do with these permissions, so it is difficult for a user to make an informed decision as to whether to allow installation.

In order to address this problem, we have developed AQUA (Android Query Analyzer), which applies static analysis to Android application binaries in order to reverse engineer aspects of the application's interaction with the user's databases. We have used AQUA to analyze 105 popular applications from Google Play [2], demonstrating that it can provide more detailed information about application behavior than is available in the application's manifest. In addition, AQUA facilitates cross-application analysis, allowing identification of advertisement libraries and of databases that are created by one application and used by another (which could obscure application access to user's private information.)

To illustrate AQUA's capability, consider the program excerpt shown in Figure 1. The manifest (line 3) declares that the app reads the user's contacts. The `getData` method (lines 7-23) displays one of two different result sets because two different content provider names are built along two different execution paths (lines 12-16). There is a string assignment for the prefix for the content providers in line 10. In case of taking the `if` branch, a built-in contact provider's name, `content://com.android.contacts/contacts`, is assembled by the concat operation (line 13). If the other branch is followed, the custom content provider name `'content://com.example.data'` is assembled (line 15). Then one of these content providers is associated with the query API to retrieve the `display_name` attribute (of a database table associated with one of the content providers) at line 19. Examining the string literals in the program in isolation would yield fragmented information, {`'content://com'`, `'android.contacts/contacts'`, `'example.data'`}, but wouldn't show how they are used to build the names of content providers used at the query execution in line 19. To do this, we need to consider how string variable values are modified.

AQUA uses a fairly lightweight flow-sensitive intra-procedural analysis to estimate the possible values of string variables that are used either directly or via content provider methods to construct SQLite queries. Using dedexer [3] to disassemble the dex binary code (the format in which Android apps are downloaded), AQUA constructs a call graph and a control flow graph slice representing relevant aspects of the program. A (slightly modified) worklist algorithm is used to associate with each node the set of possible values of string variables at that node. Since the set of possible values for a string variable is potentially infinite, a standard iterative algorithm is not guaranteed to reach a fixed point and terminate. We hypothesize that for most applications and for the variables of interest here, the algorithm will reach a fixed point after a

1095-1350/91 $25.00 © 4891 IEEE
DOI 10.1109/WCRE.2012.49

```
1  <manifest package="com.example"
2  ...
3      <uses-permission android:name="
           android.permission.READ_CONTACTS"
           />
4  ...
5  </manifest>
6
7  public class inquiry
8  {
9    public void getData(boolean param){
10     String tCP = "content://com.";
11     ...
12     if(param){
13       tCP = tCP.concat("android.contacts/
             contacts");
14     }else{
15       tCP = tCP.concat("example.data");
16     }
17     mUri = Uri.parse(tCP);
18     String mAttr[] = {"display_name"};
19     Cursor cCursor = getContentResolver()
           .query(mUri, mAttr, null, null,
           null);
20     if (cCursor != null){
21       //Display results of cCursor
22     }
23 }
```

Code 1. Motivation example

reasonably small number of iterations. This was indeed the case for over 95% of the applications analyzed.

The remainder of this paper is structured as follows. Section II gives background about Android applications. Section III describes AQUA's analysis techniques. An evaluation of AQUA's accuracy and performance and results of analyzing the top 105 apps are discussed in Section IV. Conclusions and directions for future work are presented in Section VI.

II. Background

Android is a complete package of software for mobile devices [4], [5]. It consists of four components in a hierarchy. At the bottom, there is a Linux kernel, which provides core system services such as security, memory management, process management, network connectivity, and driver models. The next layer is a combination of two components, the *Dalvik* virtual machine (DVM) for executing Dalvik Executable format (*.dex*) files and a set of core libraries for system, graphics, database, and web browsing. The third component is an application framework providing a set of services and systems fully supporting rich application development. Above all the components, there is a set of basic applications such as a phone application included in Android SDK.

A. dex code

Dalvik Virtual Machine is a register-based machine. A frame consists of a specific number of registers, a pointer to a .dex file, and data for execution. It employs a sand box architecture in which an application has its own process running on an instance of DVM. The advantage of the architecture is that a failure of an application does not impact other applications or the system. An Android application is written in Java and translated into Dalvik executable (dex) format. During translation to dex, separated and indexed constant pools and methods are derived from original Java classes. The pools store strings, types, fields, and methods. At runtime, the constant pools can be referenced by instructions as needed. Arguments to a method correspond to registers in the method invocation frame. Some assignment and array operation instructions perform for more than one type. For example, a 32-bit move instruction works for both integer and floats. Instructions are variable length.

B. Application components

There are four different types of application components used to implement an Android application. Each component has distinct purpose and conditions.

Activities
: An activity controls a single window connecting to a user interface.

Services
: A service runs in the background to support long-running operations. It does not provide a user interface but can be started by another component.

Broadcast receivers
: A broadcast receiver is a component that receives and responds to messages from other applications.

Content provider
: A content provider provides a way of accessing data repositories on the phone for both retrieving and manipulation. To use a content provider, it has to be associated with a designated API. A basic content provider class is bundled in the Android SDK and provides services for built-in data repository such as *text messages, audio, video, images, contact information*, and so on. Application developers can also define custom content providers; these often connect to private data repositories such as a table in *SQLite* database. This paper focuses on reverse engineering dex code in order to discover how applications access databases, either direct SQLite queries [1] or via content providers.

C. The Manifest File

Each application must have a *manifest*, which includes essential information describing the application's components. It

[1]direct SQLite queries are often used to create private data repositories and to manipulate data in the repositories. An app is not allowed to access SQLite database of other apps via direct SQLite queries. Note that direct SQLite queries are associated with APIs other than ones for content providers

includes needed permissions, the minimum API level required, hardware and software features used, and referenced API libraries. The manifest file, written in XML format, is bundled along with the .dex files in the application's *.apk* file. When developers distribute applications, they post the .apk file on Google Play or another repository. Consequently our analysis can only take .apk files as input, not Java source.

D. Permissions

In the Android system, applications require permission to perform sensitive operations. For example, an application needs permission in order to invoke a content provider to access data created by other applications or to access built in content providers. An application that needs additional functions not provided by default, must list the needed permissions in the manifest file. When a user installs the application, the Android system will ask for the user's consent to grant the permissions listed in the manifest. If the user denies consent, the application is not installed. Applications can create custom data repositories and make them available to other applications by declaring them in the manifest.

III. String Analysis

The goal of AQUA's analysis is to determine how the application being analyzed interacts with the user's private data stored on the Android device. To that end, AQUA estimates the possible values of strings representing SQLite queries and arguments to content provider methods that access databases. We anticipate using AQUA to create an on-line repository of information about Android apps, which users can consult when deciding whether to install an app. The analysis needs to be efficient enough to keep up with the large number of applications that are offered to users, and must have relatively few false positives (identifications of database interactions that cannot actually happen) and very few, if any, false negatives (missing interactions that can take place.)

AQUA builds on the open source dex disassembler *Dedexer* [3], which parses the dex files of the application under analysis. AQUA adds modules to build a control flow graph (CFG), to optionally perform program slicing and to analyze string variables using a form of data flow analysis. Fig. 1 summarizes AQUA's architecture.

In order to estimate the possible values of string variables, we use a slightly modified data flow analysis approach. AQUA begins by constructing a control flow graph for each method. To each control flow graph node, we associate a representation of relevant aspects of the program's state. In particular, we wish to determine the possible values of registers (dex variables) that represent string variables whenever control reaches a given node. We associate each node n with a map $VALS_n$ whose keys are registers and whose values are sets of strings. If $VALS_n[r] = \{s_1, \ldots, s_m\}$, then register r may have one of the values s_i when control reaches node n.

As described below, we compute $VALS_n$ using a slight modification of a standard work-list data flow algorithm. If the program has no operations on string variables occurring within

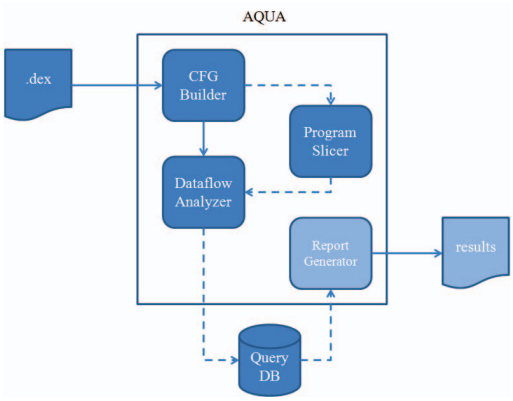

Fig. 1. AQUA Architecture Overview

loops then the algorithm will terminate with each register r mapped to a superset of the possible values of r. However some of the strings may be false positives corresponding to unexecutable paths through the program. If there are string operations within loops, the standard algorithm may not reach a fixed point and thus will not terminate. To address this we add some limits to force the algorithm to terminate. This can potentially lead to false negatives, but our experiments indicate that in the context of the variables of interest this is not usually a problem.

We compute $VALS_n$ iteratively, using a forward data flow analysis style algorithm [6]. A particular data flow analysis consists of several fundamental elements: a representation of a relevant abstraction of the program's possible states at each node, a *Transfer Function* indicating how the node's instructions modify that abstract state, and a *Join* operation indicating how states resulting from different paths are combined. The representation of the abstract state is refined iteratively as follows: sets *IN* and *OUT* are associated with each node; the values of *IN* are obtained by joining the *OUT* values of the node's predecessors (in the case of a forward algorithm) and values of *OUT* are refined by applying the transfer function to *IN*. A worklist is maintained by adding successors of any node whose *OUT* set has changed. If the problem corresponds to a partial order on a finite lattice, the algorithm will eventually reach a fixed point, where the *OUT* sets don't change further, so the worklist becomes empty and the algorithm terminates, with the final values of the *OUT* sets as results. In our case OUT_n is $VALS_n$.

Unfortunately, in our case the *OUT* sets represent possible values of string variables, and a variable could have infinitely many values. In this case, a fixed point will not be reached. To deal with this, we parameterize the algorithm with a limit *L1* on the number of times a node can be added to the work list. In addition, even when finite, the sets can potentially grow quite large, exceeding available memory. To deal with this we impose a limit *L2* on the set size. Implications of these limits are discussed below and are evaluated in Section IV.

The join operation: The *IN* and *OUT* sets of n are

TABLE I
SELECTED DEX INSTRUCTIONS AND THE TRANSFER FUNCTION

Inst. Type	Instruction	Action	Action for *P* in the transfer function
1	`const-wide/16(r, s)`	Move *s* into *r* ,where *s* is sign-extended	*OUT*(*P*) = *IN*(*P*),
	`const/4 (r, s)`	Move *s* into *r* ,where *s* is sign-extended	*OUT*(*P*)[*r*] = *s*
	`const-string (r, s)`	Move *s* into *r* ,where *s* is string	
	`aget (r, s, i)`	Get *s*[*i*] and put the value into *r*	
2	`aput (r, s, i)`	put *r* into *s*[*i*]	*OUT*(*P*) = *IN*(*P*) ,
	`move (r, s)`	Move *r* into *s* ,where *r* is contents of a non-object	*OUT*(*P*)[*s*] = *r*
	`move-object(r, s)`	Move *r* into *s*, where *r* is contents of a object	
3	`move-result(r)`	Move *n* into *r* ,where *n* is the non-object result of the most recent instruction	*OUT*(*P*) = *IN*(*P*), *OUT*(*P*)[*r*] = *n*
4	`invoke-virtual(java.lang. StringBuilder.append, r, s)`	Append *r* to *s*	*OUT*(*P*) = *IN*(*P*), *OUT*(*P*)[*s*] = *PSC(s,r)*
5	`invoke-virtual(java.lang. String.concat, r, s)`	Concatenation *r* and *s*	*OUT*(*P*) = *IN*(*P*), *OUT*(*P*)[*s*] = *PSC(r,s)*
6	`new-instance(r, t)`	Construct an instance *r* of *t* type	*OUT*(*P*) = *IN*(*P*), *OUT*(*P*)[*s*] = *r*

Initialize WORKLIST with the start node of *l*;
while *WORKLIST is not empty* **do**
  t = *remove* top node from *WORKLIST*;
  VISIT(*t*) = *VISIT*(*t*) + 1;
  if *VISIT(t)* $\leq$ *L1* **then**
    `old_OUT` = *OUT*(*t*) ;
    IN(*t*) = ϕ ;
    foreach *predecessor p of t* **do**
      IN(*t*) = JOIN(*IN*(*t*), *OUT*(*p*));
    end
    OUT(*t*) = Transfer(*IN*(*t*));
  end
  if *OUT(t)* $\neq$ `old_OUT` **then**
    foreach *successor s of t* **do**
      Add *s* to *WORKLIST* ;
    end
  end
end

Algorithm 1: Data flow analysis

approximations of $VALS_n$, i.e., they are maps from registers to sets of strings. The *IN* value of a node is based on the *OUT* values of its predecessors, using a join operation that unions the values of each register:

$$IN(n)[r] = \bigcup OUT(p)[r]$$

where the union is taken over all predecessors p of n. In other words a string s is a possible value of r coming into node n if s is a possible value of r coming out of *any* of n's predecessors.

The transfer function: In general a transfer function computes *OUT* of a node based on *IN* of the node and the semantic actions performed by the instructions the node represents. The transfer function is based on Dex instructions relevant to string operations. Table I describes the transfer function in detail. The current implementation takes account of the most frequently used string operations.

By examining dex code of applications, we found that there are two kinds of dex instructions that have to be considered. First, there are instructions that directly manipulate string registers, e.g. by assignment. The transfer function for assignment operators updates the map by replacing the possible value set of the left-hand side by the possible values of the variable on the right hand side.

Second, there are calls to relevant string library functions. AQUA currently handles string concatenation operations. The Java instruction `z = x.concat(y)` is translated to dex instructions corresponding to `invoke-virtual java.lang.String.concat(x,y)`, followed by `move-result-object(z)`. Similarly, `z = x + y` in Java is translated `invoke-virtual java.lang.StringBuilder.append(y,x)` followed by `move-result-object(z)` in Dex. In both cases, the first invoke-virtual instructions generates new elements then the second instructions assign the results of the operation to the target registers.

The transfer function checks invoke-virtual instructions to see whether they are invocations of `concat` or `append` and, if so, updates the possible values of the target string register using a *Pairwise String Concatenation* (*PSC*) on the sets of possible string values of the arguments. *PSC*(*R*,*S*) is the set obtained by concatenating each element of *R* with each element of *S*. For example, suppose *R* = $\{ r_i \mid 1 \leq i \leq n$ *and* r_i is string $\}$ and S = $\{ s_i \mid 1 \leq j \leq m$ *and* s_j is string $\}$ Then,

$$PSC(R,S) = \bigcup_{i,j=1}^{i,j=n,m} r_i s_j$$

In examining the application code, we also noted that arrays of strings were quite common. We model arrays of strings as sets of strings, ignoring the indices. For example, consider the dex instruction `aget(r, s, i)`, which fetches element *i* of array *s* and assigns its value to *r*. We represent the possible values of all elements of *s* with a single set; the transfer function assigns this set to be the possible values of *r*. Similarly, assigning a string `r` to an array element *s*[*i*] adds the possible values of `r` to the set representing possible values of any element of *s*.

Worklist Algorithm: Algorithm [1] summarizes the string analysis algorithm. The inputs for the algorithm are *CFG*s, a

limit for loop iterations *L1*, and a limit *L2* for set size. Each node includes attributes *IN, OUT*, and *VISIT*. *IN* and *OUT* are hash maps in which the *keys* are *registers* that can represent strings at the given node and the *values* are sets of possible string values for the register. Initially all *OUT* maps are empty. If the transfer function modifies the *OUT* map of a node, the node's successors are added to the worklist, unless that node has already been visited *L1* times. In addition the transfer function does not add elements to the value set of a register if the size exceeds *L2*. When an invocation of interesting query APIs such as `getContentResolver().query()` is encountered, the possible value sets associated with the API call are stored in a *SQL* database. The report generator uses the database to generate reports in html.

Consider our algorithm on the CFG in Fig. 2. Since there is an assignment statement in *P*, *OUT*(*P*)[tCP]={'content://com.'}. *OUT*(*P*) then becomes *IN*(*Q*) and *IN*(*R*). The result of concatenation on `tCP` at *Q* is `'content://com.android.contacts/contacts'` so that the possible value of `tCP` in *OUT*(*Q*) is replaced by {`'content://com.android.contacts/contacts'`}. Similarly, the possible value of `tCP` in *OUT*(*R*) is replaced by {`'content://com.example.data'`} Since *Q* and *R* are the predecessors of *S*, *IN*(*S*) = JOIN(*OUT*(*Q*) , *OUT*(*R*)). Thus, *IN*(*S*)[tCP] = *OUT*(*S*)[mUri] = {`'content://com.android.contacts/contacts'`, `'content ://com.example.data'`}. Note that `Uri.parse(tCP)` is considered as an assignment instruction. The possible values of `mUri` in *IN*(*U*) are associated with the query API in *U* because *T* does not change the possible values of `mUri`. Thus, there are two sets of data available from the query, one the data repository linked to the content provider `'content://com.example.data'` and the other one from `Contact`. In addition, AQUA infers that the second argument of the query method is `'display_name'`, indicating that this attribute is fetched from one of the content providers by the query.

false negatives: If the application being analyzed has string operations within a loop, it is possible that $VALS_n$ will be infinite for some nodes and registers. For example, consider a program that appends strings to a variable v within a loop. Even if the loop only can iterate finitely many times, the iterative algorithm will consider the infinitely many paths through the loop and continue to derive new possible values for v. In this case a standard iterative algorithm will not terminate. We force termination, by keeping track of the number of times each node t is added to the work list *VISIT*(t) If this counter reaches *L1* the node's successors are not added to the worklist.

In addition, the pairwise concatenation of sets of strings can result in exponential growth in set size, even when there are no loops. This occurs in programs with a lot of conditional jumps. To avoid running out of memory, a second limit is used. If the set of values for a register grows beyond *L2* the new elements are not added to the set.

Both of these limits can lead to false negatives, i.e., to possible values for a string variable that are not found by

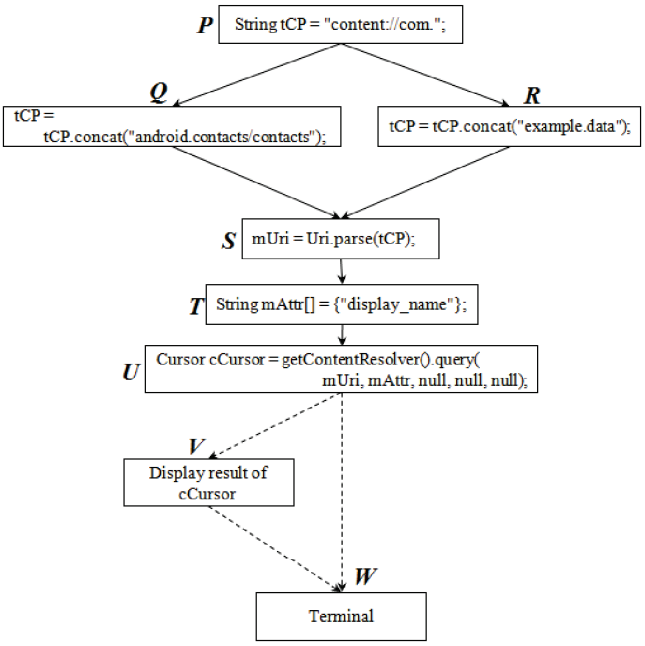

Fig. 2. CFG of `getData` in the motivation code

AQUA. As shown in the Section IV this is not a problem for most programs. Furthermore AQUA can indicate whether a limit was reached, so that users will know whether the results are completely trustworthy. More cautious users may opt not to install an application for which AQUA reports a potentially incomplete analysis.

Alternative approaches to string analysis that are guaranteed to be safe exist, but there is the potential of more false positives. Future research will explore this trade-off more fully.

program slicing: We expect that applications will modify strings that are used to build SQLite queries (directly or via content providers) within a loop will be rare. However, it is not unusual for applications to modify other string variables within a loop. In particular, applications often iterate through an array or a result set returned by a query, concatenating the contents onto a string that will eventually be output. Such constructs can cause our algorithm to terminate early (hitting L_1). The problem can be alleviated by computing a program slice [7] that only includes nodes that are potentially relevant to the query execution points of interest.

In the current version of AQUA, we used a lightweight program slicing technique based solely on control flow. A depth first search is performed, marking nodes n for which there is a path from n to a query execution call. Other nodes are eliminated and the edges are adjusted accordingly. In the example, nodes V, W and the edges to them are eliminated. As discussed in Section IV, this somewhat improved AQUA's performance. Adding a more sophisticated slicing algorithm may further improve the results.

Sample AQUA result: Fig. 3 shows the result of running AQUA on an application from Google Play. The result gives

Information	#Use	Used Attribute
bookmark	0	
call log	0	
audio	0	
mms	0	
contact	5	primary_email name number type display_name mimetype contact_id
video	0	
Private data repostitory	1	install_referrer

Fig. 3. Actual result on a tool app that uses `Call Log` and `Contact` 3 times and creates a private data repository

names of built-in data repositories, custom databases, attributes associated with built-in or custom data repositories and the number of times data was accessed. Note that the Manifest or program description would not explain what kind of custom databases are created and/or used, which attributes are used, or which databases are accessed via direct SQLite queries.

IV. Evaluation

To evaluate our approach in terms of performance and accuracy, we ran AQUA on the top 105 free applications from Google Play, observing values of variables used in accessing built-in and custom data repositories. We used a Windows 7 machine with Intel i7, 2.67 Ghz CPU, and 8 GB memory for the measurements.

A. Analysis Results

Recall that we are interested in determining the parameters to content provider API calls and to SQLite API calls. The method being called at a given invoke instruction is determined (by dedexer) by referring to the method pool. For content provider methods, the first parameter is a string representing the URI of the content provider. If the set of values for this parameter has k elements, then k different data repositories could be accessed. We consider each of these to be a different *Query Call Site (QCS)*. For example, node *U* in Fig. 2 has 2 QCS's. In addition to the URI, content provider methods have parameters corresponding to the names of database tables and attributes being accessed, the conditions under which a row is returned, etc. (In the example, several arguments are null, representing default values.) We define *Query Site Values (QSV)* to be the possible combinations of parameter values at a query call site. For example if there were three possible values of the second argument and five possible values of the third argument in addition to two URIs, there would be $2 \times 3 \times 5 = 30$ query site values, representing 30 different database queries that could be executed at that point.

Some of the parameters to a content provider method have further syntactic requirements. For example the first parameter must represent a URI. On examining AQUA's results, we noted that some strings are not syntactically correct. These are false positives representing strings that are constructed along infeasible paths (or, less likely, they are strings that would lead to run-time failures.) We classified the sets of values of arguments to relevant methods into four possible types. A set is *Complete* (*CO*) if all elements in the set are syntactically correct. For example, {`'com.android.contracts/contracts'`,`'content://mms'`}. A set is *Bad* (*B*) if all elements in the set are syntactically invalid. For example, {`'//content://'`}. A set is *Incomplete* (*INC*) if all elements in the set are invalid, but can be used to produce an valid value such as {`'//'`}. A set is *Mixed* (*MI*) if it combines the three types mentioned earlier.

Manual inspection of AQUA's result on a set of sample applications showed that it correctly identified all QCS from sample applications that included 25 query API call sites. The reason for using the sample application is that source code of an application is not distributed in general. The first application had 9 call sites and various execution paths including if, if-else, nested if-else, and combination all of them. The other two applications included 17 query call sites using built-in or custom content providers. We ran AQUA on these applications setting *L1* = 200 and *L2* = 20. The result showed that the analyzer correctly explored 25 QCS. We observed set types in the result on the sample applications. There were 75 possible value sets associated with the injected QCS. 73 sets were in *CO* (97%), two sets were *MI*, and one set was *INC*.

We also manually examined 605 possible value sets at query API call sites in one of the subject applications from the Google Play, setting *L1* = 200 and *L2* = 20. In terms of number of elements in a set, 85% have one element, 8% have 2 elements, and 7% have more than 2 elements.In terms of combination of set type and number of elements, 65% are in *CO* type having one element, 20% are in *INC* type having one element, 6% are in *CO* type having two elements, and 3% are in *CO* type having more than two elements. Increasing *L2* to 200 did not change the results. This shows that many queries were associated with specific possible value sets that have one or two element(s) in *CO* type and increasing *L2* would not affect type and number of elements in the result.

1) Analysis Results for Google Play apps: Table II summarizes the 105 apps downloaded from Google Play, showing the number of applications in each category, along with the average number of QCS, QSV, and content providers found by AQUA. AQUA discovered 679 content provider uses and 1,261 direct SQL query execution. Note that numbers of content provider uses were explored by counting query API calls that were associated with *CO* or *MI* type value sets for content provider arguments. Further note that strings representing content provider can be written in two ways. One is an explicit content provider that start with `'content://'`. For example, `'content://contact'`. The other one is a helper class that contains an explicit content provider. For example, `'android.provider.Contacts.Phones'`

TABLE II
NUMBER OF QCS, QSV, CONTENT PROVIDER EXTRACTED WITH *L1* = 200 AND *L2* = 20

Category	# apps	Avg. QCS	Avg. C.P.	Avg. QSV
Books	2	53	0	63
Communication	17	99	19	122
Entertainment	7	24	2	39
Finance	1	0	0	0
Game	28	15	1	19
Media & Video	2	18	3	20
Music & Audio	9	60	2	71
Personalization	3	221	20	291
Photography	4	31	5	41
Productivity	5	36	5	45
Shopping	2	9	1	9
Social	8	123	9	185
Sports	1	100	1	100
Tools	12	42	4	45
Travel & Local	2	57	16	73
Weather	2	57	5	60
Grand total	105	54	6	70

TABLE III
CONTENT PROVIDER USE IDENTIFIED WITH *L1* = 200 AND *L2* = 20

C.P.	# use
Browser Bookmark	7
Calendar	1
Call Log	12
Contact	354
Image	10
MMS	14
Video	4
Program	244
Grand total	646

437 content provider uses were built-in (64%). Most of built-in content providers were *Contact* (354/81%). We also found that applications in *Communication* category used more built-in (249/57%) content providers than others.

On the other hand, 244 content provider (43%) uses were custom content providers declared in the applications. These custom content providers most likely used private data repositories such as SQLite tables. In addition, applications in *Communication, Personalization*, and *Social* used more custom content providers (154/63%).

2) Third-party library and private data use: Surprisingly, we found four game applications that used *Contact*. One of the games was even capable of manipulating contact information via the content provider. Further examination showed that the query API calls for Contact were located in the class `com.mobclix.android`. The class was most likely bundled in the third party SDK for advertisement *Mobclix* [8]. We suspect that the game application uses Contact information for advertisements.

AQUA's results can also be used for cross-application analysis. We identified 14 third-party libraries capable of using private data such as contact, that were used in more than one application. There was an advertisement library that was referenced by 19 different applications. Also, Mobclix was used in 8 different applications. There are several advertisement libraries [9]–[11] available and we expect AQUA to be useful for determining which of these (or others) are used by which apps.

It would be possible for an application to obtain the user's sensitive information (e.g. Contact info) from a built-in contact provider (with permission), then to store it in a private data repository, without the user's knowledge. Other applications could then access this provided data repository via a custom content provider, thereby obscuring the leak of sensitive information. AQUA's results over a set of applications can be examined to look for multiple applications that use the same custom content provider.

B. Effects of limits

1) reaching a fixed point: Recall that our algorithm is not guaranteed to reach a fixed point and therefore we add a limit *L1* on the number of iterations. We hypothesized that this would not be a problem for the string variables of interest. To check this we analyzed all the applications with *L1* = 200 and *L2* = 20 and noted the maximum number of times that a node was added to the worklist for each application. (As long as this is less than *L1*, the analysis terminates normally.) On most applications the analysis algorithm terminated after a small number of iterations. 96 applications (91%) terminated in fewer than 10 iterations. There were three applications that iterated more than 25 times and no application hit the limit *L1*.

In many cases, program slicing helped to reduce the number of loop iterations. To measure effectiveness of the program slicing, we ran AQUA on the 105 subject applications with the program slicing on and off, recording the number of iterations performed on each application reached before termination. Fig. 4 shows the number of applications that terminated in at most x iterations with slicing on and off. With program slicing on, 88 applications (71%) terminated within three iterations; while 71 applications (67%) terminated within three iterations with program slicing off. For the applications that required more than 9 iterations, the behavior (number of iterations before termination) was the same with slicing on or off. Recall that our current slicer only used control flow information. In future work, we will explore the effect of more powerful slicing techniques.

Next, we explored the effect of *L2*. With program slicing on, there were five applications for which the number of iterations (i.e., the maximum number of times a node went onto the worklist) increased when *L2* increased; [2] the remaining 100 applications terminated having the same numbers of iterations regardless of *L2*. As shown in Fig. 5, for *L2* = 50, each of the five apps terminates after about 50 iterations and for *L2* = 200, all of these five applications hit the limit(*L1* = 200). This happens because *L2* can hide some differences in *OUT* sets, thereby causing nodes not to be added to the worklist, even though the actual set of values of some string variable has changed. By inspecting their dex code, we found that there was at least one string concatenation included in a loop in each of

[2]Without program slicing,six apps exhibit this phenomenon.

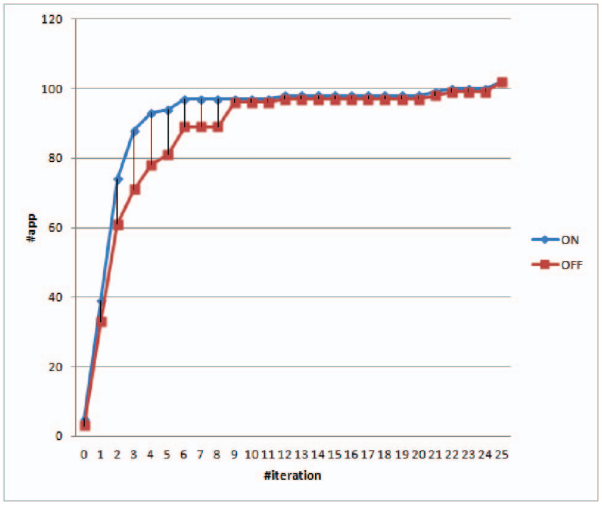

Fig. 4. Number of iterations (max = maximum number of times any node goes onto the worklist before termination) with the program slicing on and off. The X-axis represents number of iteration and the Y-axis represents number of applications for which $max \leq x$. There was also one application that terminated after adding a node 41 times onto the worklist. AQUA was run with *L2* = 20.

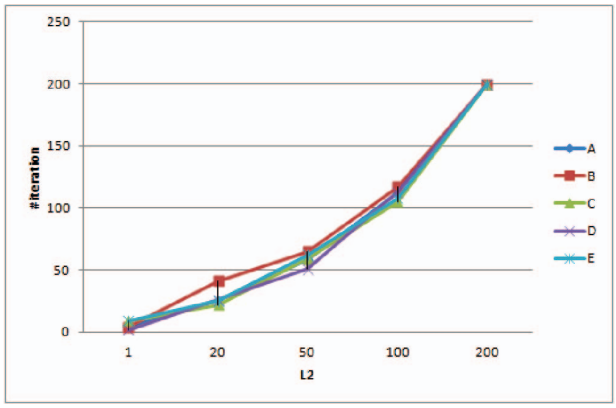

Fig. 5. Five applications for which AQUA reaches *L1* with large *L2*. The X-axis represents *L2* and the Y-axis represents max number of times any node goes onto the worklist.

these programs. It is possible that AQUA misses important information (i.e., has false negatives) on these anomalous programs, but in these cases, AQUA can report that its results are potentially incomplete. In future work, we will explore more powerful techniques to use on such programs.

2) number of Query Call Site: By analyzing results of AQUA, we found that the numbers of QCS extracted did not change much with different *L1*. For *L1* = 1, 5,692 QCS were identified. We also increased *L1* = 10, 50, 100, and 200 but 5,697, 5,699, 5,697, and 5,697 QCS were extracted respectively. Moreover, in terms of number of applications that gain more QCS with increased *L1*, 5 applications extracted more QCS with *L1* = 10 than *L1* = 1; 2 applications extracted more QCS with *L1* = 50 than *L1* = 10. After *L1* = 50 numbers of extracted QCS were the same. Note that we set *L2* = 20 and

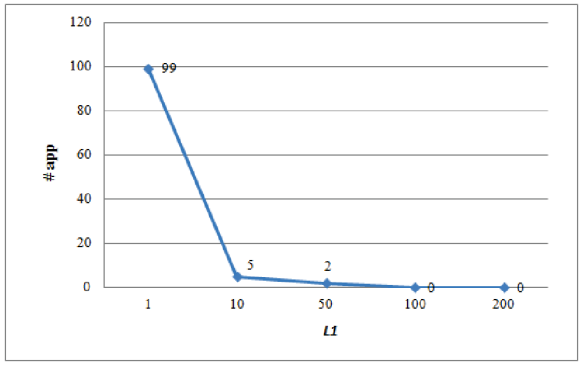

Fig. 6. Numbers of applications that gained more QCS with (*L1*+1) than *L1* while *L2* = 20, X-axis represents number of application and Y-axis represents *L1*, For *L1* = 1, #app is 99 because there are 6 apps having no QCS

obtained the total QCS by counting query call sites that have different possible value sets at each call site. It is summarized in Fig. 6

We investigated the average number of QCS used in applications in each category. There were 54 QCS per app on the average. The highest QCS was 359 with a personalization application while one finance application has no queries. In terms of average number of QCS in category, applications in Personalization use about 220 QCS. In terms of query site value, AQUA identified 7,407 QSV (70 QSV/app). The personal application has highest QSV 736 again. Table II summarizes.

C. Performance

On the average, AQUA processed an application in about 20 sec using about 1GB memory and its performance was not affected by increasing *L1*. For *L1* = 1, the analyzer took about 22 sec/app and used about 1.1 GB/app. The longest execution time was 115 sec. The largest memory use was about 2.5 GB. For increasing *L1* = 200, the average execution time and memory use slightly increased to 23 sec/app and 1.5 GB/app. The longest execution time was 130 sec and the largest memory use was about 3 GB.

V. Related Work

There are several approaches that apply static analysis to mobile applications. Chaudhuri proposes a formal model [12] that traces flows between applications referring to permissions. In follow-on work, Fuchs et al. propose SCanDroid [13], which identifies permissions for content providers from an application's manifest, analyzes content provider uses, and checks whether data flow through identified content providers violates the extracted constraints. Both ScanDroid and AQUA use data flow analysis on content providers. ScanDroid decompiles dex code into Java byte code of the applications and uses Java analysis tools, while AQUA works more directly with the dex code. While ScanDroid provides information about which content providers are used, AQUA provides more detailed information about how content providers are used, along with

information about direct SQLite query uses. Applying the ScanDroid formal model would give AQUA a capability of analyzing constraint violation.

Egele et al. propose PiOS [14], which constructs control-flow graph for iOS application binaries to check whether information leaks are present by using data flow analysis. They found a number of applications on official App Store and third-party repository that leak sensitive information.

Chin et al. propose ComDroid [15], which analyzes disassembled DEX bytecode to look for vulnerabilities in message passing via Intent between applications. It also employs CFG construction and data flow analysis but does not explicitly work on content provider use. Felt et al. propose Stowaway [16], which provides a mapping of API calls to permissions and identifies over-privileged applications. It reports content providers that appear as single string literals, but does not examine other content provider API arguments or deal with string concatenation.

Enck et al. propose ded [9] that decompiles Android dex code to get corresponding Java code. By using ded they provide insight into how Android applications behave. However, they don't provide much information about content provider use.

There are several dynamic analysis tools for mobile applications. Enck et al. propose TaintDroid [17] that identifies that applications send privacy sensitive information to network servers using dynamic taint analysis. It found a number of potential information misuse from example applications.

The underlying problem addressed by AQUA is that Android users don't have enough knowledge or control over how apps use sensitive data. While AQUA addresses this by providing more detailed information about app behavior, informing users decisions about whether to install an app, several recent works work dynamically to prevent apps from accessing sensitive data. TISSA [18], proposed by Zhou et al., and MockDroid [19], proposed by Beresford, supply fake information when there is a request for sensitive data from untrusted applications. AppGuard [20] modifies dex binaries so that security policies are checked and exceptions are thrown when they are violated. Dynamic approaches could be useful for checking AQUA's precision. On the other hand, AQUA's analysis could potentially help in targeting the dynamic application of security policies.

There are many approaches for enhanced security policy. Enck et al. propose Kirin [21], which prevents the installation of applications that request specific combinations of permissions that allow malicious actions. Ongtang et al. propose Saint [22], which provides policies that allow application developers to declare install-time and runtime constraints on permission uses. Ongtang et al. also propose Porscha [10], which enforces digital rights management policies for content. Porscha mainly applies to Email, SMS, and MMS that associated with built-in content providers.

The prevalence of string operations in web applications and database applications has led to a lot of recent research on string analysis. Java String Analyzer [23] statically analyses Java byte code to derive automata representing possible values of string variables. This has been applied in several ways to analyze SQL queries in Java/JDBC programs [24], [25]. Yu et al [26] apply data flow analysis using a compact automaton representation of string variable values. In contrast to AQUA, these techniques are *safe* in the sense that they are guaranteed to derive a superset of the possible values of each string variable (i.e., there are no false negatives). Yu achieves this by using a data flow algorithm, but introduces a widening operator to deal with loops. While avoidance of false negatives is necessary in some contexts, it can lead to overly conservative results which have many false positives. In the context of AQUA, we believe allowing some false negatives and warning users that they may be present is an appropriate approach. In future work, we will explore whether more compact representations of sets of strings significantly enhance AQUA's performance.

VI. Limitation and Conclusions

AQUA uses a fairly lightweight static analysis to reverse engineer Android applications' dex code in order to explore how the application interacts with built-in and custom data repositories on the user's phone or tablet. We evaluated AQUA on 105 popular Android applications and found that it produced useful and quite precise results. We anticipate using AQUA to provide an on-line service in which applications are analyzed and detailed information about their data use is posted. This will allow potential users to make more informed decisions about whether to install an app.

There are several ways that we could improve precision of our analysis. As shown in result section, there were a number of INC value sets. Some of them arise from inter-procedural calls. There were numbers of variables that missed their possible values. They were most likely variables whose values were assigned outside of the method under analysis. We expect that plugging in inter-procedural analysis of selected methods may efficiently remove numbers of *INC* or *MI* value sets and discover missing possible values.

One limitation of the current implementation is that some relevant instructions may be overlooked by the transfer function, especially during selecting the main instruction from a series of related instructions. This limitation could be easily fixed by modeling the additional instructions.

There are still two structural limitations inherited from data flow analysis, under and over estimation. Over estimation (false positive) occurs when string values that could only be constructed along infeasible paths are found. This is an inherent problem with the approach. Because we are applying data flow analysis to a problem that corresponds to an infinite lattice under-estimation also occurs, though the evaluation shows that this is rare. This happens when aborting loop iterations before reaching substantial result sets or when limiting sizes of sets of values.

Despite these limitations, AQUA's analysis is quite precise and we expect it to be a useful tool for protecting Android users' privacy.

In future work, we will explore using more compact representations of the set of possible string values (such as automata). This will eliminate the need for *L2* and we expect it will also allow larger values of *L1* to be practical, if needed. We also plan to explore whether using more sophisticated slicing algorithms substantially improves AQUA's precision and performance. To automatically determine whether an app maliciously uses private information of the user we will explore threat models characterizing apps that access private data and store it in other data repositories or transmit it to other hosts or apps. We will offer the information AQUA produces to real users to see how the information helps to make a decision on the installation.

ACKNOWLEDGMENT

This work was partially supported by National Science Foundation grant CCF-0541087.

REFERENCES

[1] N. Perlroth and N. Bilton, "Mobile apps take data without permission," *NY Times*, 2012. [Online]. Available: http://bits.blogs.nytimes.com/2012/02/15/google-and-mobile-apps-take-data-books-without-permission/

[2] "Google play," https://play.google.com/store/apps.

[3] "Dedexer," http://dedexer.sourceforge.net/.

[4] "Android developer's guide," http://developer.android.com/guide/index.html.

[5] "Android sdk," http://developer.android.com/index.html.

[6] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, pp. 137–, mar 1976.

[7] F. Tip, "A survey of program slicing techniques," *JOURNAL OF PROGRAMMING LANGUAGES*, vol. 3, pp. 121–189, 1995.

[8] "mobclix," http://www.mobclix.com/.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

[10] M. Ongtang, K. Butler, and P. McDaniel, "Porscha: policy oriented secure content handling in android," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 221–230.

[11] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *ICISS*, 2011, pp. 49–70.

[12] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09. New York, NY, USA: ACM, 2009, pp. 1–7.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications."

[14] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[15] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.

[17] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[18] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proceedings of the 4th international conference on Trust and trustworthy computing*, ser. TRUST'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 93–107.

[19] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11. New York, NY, USA: ACM, 2011, pp. 49–54.

[20] M. Backes and S. Gerling, "Appguard - real-time policy enforcement for third-party applications," Tech. Rep., 2012.

[21] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245.

[22] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," *Computer Security Applications Conference, Annual*, vol. 0, pp. 340–349, 2009.

[23] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *In Proc. 10th International Static Analysis Symposium, SAS 03, volume 2694 of LNCS*. Springer-Verlag, 2003, pp. 1–18.

[24] C. Gould, Z. Su, and P. T. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of International Conference on Software Engineering*, 2004, pp. 645–654.

[25] W. G. J. Halfond and A. Orso, "Command-form coverage for testing database applications," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 69–80.

[26] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, "Symbolic string verification: An automata-based approach," in *in Proc. of SPIN, 2008*, pp. 306–324.