Transparent Privacy Control via Static Information Flow Analysis

Xusheng Xiao   Nikolai Tillmann   Manuel Fahndrich
Jonathan de Halleux   Michal Moskal

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{t-xuxiao, nikolait, maf, jhalleux, micmo}@microsoft.com

Microsoft Research Tech Report MSR-TR-2011-93

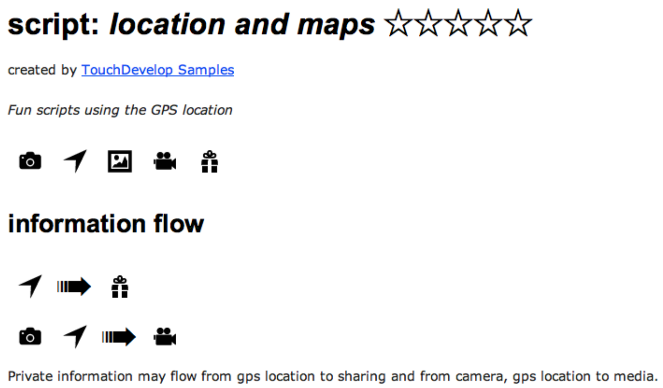
August 7, 2011

Abstract

A common problem faced by modern mobile-device platforms is that third-party applications in the marketplace may leak private information without notifying users. Existing approaches adopted by these platforms provide little information on what applications will do with the private information, failing to effectively assist users in deciding whether to install applications and in controlling their privacy. To address this problem, we propose a transparent privacy control approach, where an automatic static analysis reveals to the user how private information is used inside an application. This flow information provides users with better insights, enabling them to determine when to use anonymized instead of real information, or to force script termination when scripts access private information. To further reduce the user burden in controlling privacy, our approach provides a default setting based on an extended information flow analysis that tracks whether private information is obscured before escaping through output channels. We built our approach into TouchDevelop, a novel application-creation environment that allows users to write application scripts on mobile devices, share them in a web bazaar, and install scripts published by other users. To evaluate our approach, we plan to study a portion of published scripts in order to evaluate the effectiveness and performance of information flow analysis. We also plan to carry out a user survey to evaluate the usability of our privacy control and guide our future design.

1 Introduction

Modern mobile-device platforms like iOS [3] , Android [1] and Windows Phone [7] provide a central place, called app stores or marketplaces, for finding and downloading



script: *location and maps* ☆☆☆☆☆

created by TouchDevelop Samples

Fun scripts using the GPS location

information flow

Private information may flow from gps location to sharing and from camera, gps location to media.

Figure 1: Visualization of information flow of a sample application

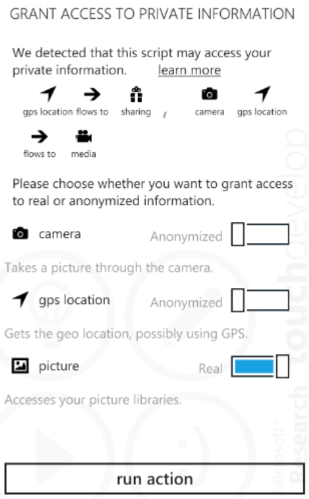
third-party applications. These third-party applications allow users to add more functionality to their devices. For application developers, the central marketplace makes it easy to distribute their apps. Applications for mobile devices are typically written using traditional programming languages in traditional desktop development environments that provide APIs to access the touch screen, sensors, network and so on.

TouchDevelop [5][1], a novel application-creation environment, enables users to write small applications directly on mobile devices using touch screens. We call applications written in TouchDevelop "scripts". TouchDevelop also provides a "script bazaar" that allows users to publish their scripts; the source code of a script is made available as part of the publishing process. Other TouchDevelop users can then discover, comment, and review published scripts. If desired, users can install published scripts onto their own mobile devices and run them as well as modify them. The TouchDevelop script bazaar is similar to the existing marketplace concept, but all scripts are free to download and install, and their source code is available as well.

A common problem faced by these mobile-device platforms is that the published applications in the marketplace may leak private information through output channels. Many of these applications access the mobile-device resources like sensors and GPS that may contain or capture and expose private information, and share them using remote cloud services or web services [11] without notifying users. Similarly, in TouchDevelop, a script can read a user's geolocation and post it on facebook [2] or silently send it to a web service. To mitigate these problems, iOS requests permissions before applications can access users' geolocations and Android asks for permissions before users can install applications. However, these platforms provide little information about how these applications may use users' private information, requiring users to make uninformed decisions on whether to install applications and how to control their privacy. This privacy control mechanism leads to a situation where users simply install applications without questioning the requested permissions, even if the applications ask for more permissions than needed [11, 12].

To better protect users' privacy while preserving the fun and utility of applications,

[1] TouchDevelop has been previously called as TouchStudio [25]



GRANT ACCESS TO PRIVATE INFORMATION

We detected that this script may access your private information.

Please choose whether you want to grant access to real or anonymized information.

camera — Anonymized

Takes a picture through the camera.

gps location — Anonymized

Gets the geo location, possibly using GPS.

picture — Real

Accesses your picture libraries.

Figure 2: Grant Access To Private Information

we propose a transparent privacy control approach, where an automated static analysis reveals to the user how private information is used within a script. The results of this information flow [9] analysis are shown to the user, thereby enabling her to make informed decisions about when to use anonymized instead of real data, or to force termination of scripts when they access private information sources, such as the phone's geolocation and camera. We use the term *Source* to refer to the origin of any private data and similarly, we use *Sink* to refer to any point where data may leak from a script. When an application script is submitted for publication, our approach automatically computes all information flow of private information via static analysis and visualizes the information flows, as shown in Figure 1. The visualization of a script's information flow shows what sources of private information the script accesses, and how that information may leak from the user's device.

The computed information flows by themselves may not be useful enough for users to make decisions whether or not to grant a script access to private information. For example, a script can encode the user's phone number into the color intensity of some pixels inside a picture to be shared. The information flow will reveal that private information from the camera and contact sources flow to the share sink, but a user may be hard pressed to recognize any changed pixels in the picture being posted.

To distinguish the sharing of a picture taken directly from the camera or the picture library from the potential malicious sharing of a tampered picture, our approach extends the static analysis to track whether private information is tampered with before it leaks from the device into a sink (e.g., the web). Using this tamper information, we can define a policy to classify an information flow as a safe or unsafe: if untampered private information flows into a *vetted* sink, which shows a dialog for users to review the information, our approach considers the flow as a safe flow; otherwise, the flow is an unsafe flow.

When the application is executed for the first time, our approach allows users to choose among *real* data (default for sources only appearing in safe flows), *anonymized* data (default for sources in unsafe flows), or *abort* execution[2], as shown in Figure 2. This anonymized/real/abort setting provides two more flexible choices for users: (1) using the anonymized information, users can experiment with applications before granting applications the access to real information; (2) aborting an execution prevents an unintended access to a resource, and leaves a stack trace at the access point for diagnosis. To minimize the extra burden introduced on users, our approach does not ask users to grant access to the private information not flowing to sinks, and provides default settings based on whether the private information only appears in safe flows.

We built a prototype of our privacy control into the TouchDevelop app and deployed the static information flow analysis as part of the cloud services that allow users to share scripts. The availability of the submitted scripts' source code and the simplicity of TouchDevelop language make TouchDevelop a suitable platform to evaluate our approach. We plan to study a portion of published scripts for evaluating the effectiveness and performance of our information flow analysis. We also plan to carry out a user survey to evaluate the usability of our privacy control and guide our future design.

2 TouchDevelop Language

TouchDevelop allows users to create applications using an imperative and statically typed language [25]. The imperative features of the language enable users to update local and global variables, as well as the state of objects. Static typing enables the TouchDevelop editor to make good suggestions for property selection and argument completion, which makes editing code easier and faster.

A TouchDevelop script consists of a number of actions (procedures) and global variables. The body of actions consists of: (1) expressions that either update local or global variables (assignments), invoke another action, or invoke a predefined property; (2) conditional statements **if**−**then**−**else**, (3) loop statements, **for**, **while**, and **foreach**, that iteratively execute a block of statements. The global variables are statically typed and their current value is accessible across multiple actions.

As a statically typed language, TouchDevelop defines a number of data types (e.g., **Number** or **String** for s, or Picture for p in Figure 3). Each data type provides a number of properties (e.g., → create picture of global singleton object $media$). For the sake of the simplicity, the language does not provide features that allow users to define new types or properties.

For the purposes of our information flow analysis, we map the TouchDevelop concepts to a simpler model. We can think of each kind of value as having two separate parts: 1) *an immutable part*, and 2) *a mutable part*. Many types of values have only an immutable part and no mutable part, e.g., **Number**, **String**, and **GeoLocation**. Other types of values have both immutable parts and mutable parts. E.g., Picture has an immutable part that is associated with whether the picture is valid (i.e., whether the pointer is null). The mutable part of a picture consists of the actual pixel colors at each coordinate of the picture.

[2] The *abort* option is not implemented in first released version of TouchDevelop.

```
action foo() : Nothing {
        var s := ”unclassified”;
        var p := media → create picture();
        var l := senses → current location; // classified
        s := l → describe(); // classified
        p → draw text(s); // p’s mutable state is classified
        p → share(”facebook”);
}
```

Figure 3: Example of classified information flow.

Mutable and Immutable Values: We track information flow separately for the mutable and immutable parts of values. The immutable part of an object is copied whenever a value is assigned from one local to another, passed as parameter, returned from a method, stored or loaded from a global variable. The immutable part of a value is tracked precisely at each program point and assignments are strong assignments that replaces the original values.

The mutable part of an object is affected only by pre-defined property invocations (i.e., primitive methods). We track the mutable part of values using an abstraction where we have a single mutable location per type. Every value of that type shares that same mutable location. All updates to the mutable part are weak updates.

Primitive properties are annotated with information that indicates from which parameters (and thus which kinds) the mutable state is read, and also which mutable states are written (parameters and return values).

Embedded References: Because values may have embedded pointers to other values that could be mutable, we also keep track of such embedded references using directed edges from one mutable location to another. The model currently does not accommodate references from immutable parts to mutable parts, but we have not found a need for that. Establishing a reference from one value to another implies a write to the mutable state of the first.

Globals: To simplify the description in the remainder of the paper, we eliminate global variables from the model completely. Global variables are treated as extra parameters and return values from each action. One can easily transform a program with globals to a program without globals by adding all globals used in an action (and actions called) as extra parameters, and all globals modified by an action as extra return values. Inside an action, access to a global is not different than access to a local variable.

In the explanations of the rest of the paper, we will thus no longer explicitly talk about global variables.

Parameters: Parameters of an action are treated as ordinary locals inside an action. They are pre-initialized by the action invocation, but otherwise act no differently than normal local variables.

Results: Result variables are ordinary locals inside an action. Upon return, their immutable parts (values) are copied to the caller's locals that receive the results of the invocation.

2.1 Simplified Language

We assume that our input program consists of a number of actions, where each action has any number of parameters and any number of results. The body of an action consists of a control flow graph of basic blocks, with a distinguished entry block and a distinguished exit block. Conditionals branching on condition c are transformed into non-deterministic branches to the **then** and **else** blocks, where the target blocks are augmented with a first instruction of the form assume(c) and assume(**not** c).

The instructions inside a block have the following forms:

$$\begin{aligned} \mathit{Instruction} ::= {} & x := y \mid r := p(x_1..x_n) \\ & \mid r_1..r_n := a(x_1..x_m) \mid \mathit{assume}(x) \mid \mathit{assume}(\neg x) \end{aligned}$$

An instruction is either a simple assignment from one local to another, a primitive property invocation of parameters $x_1..x_n$ binding the result to a variable r, an action invocation with parameters $x_1..x_m$ binding the results of the action to $r_1..r_n$, or a special assume statement arising from conditional branches.

We assume that primitive operations always return a value, even if it is the **Nothing** value.

2.2 Classified Information Flow

In this section, we illustrate several examples to show how scripts written in TouchStudio may leak private information (referred to as *classified information*). Figure 3 shows an example of how classified information flows among values, such as **Number** and **String**. At line 4, the geolocation l becomes classified since it contains the geolocation data obtained via the GPS. Here, we refer to the property senses → current location as a *Source* of geolocation data. At Line 5, the location is transformed into a string and assigned to s, thereby making s classified. At Line 6, the location string s is rendered as text into the picture p, causing p to be classified. At Line 7, the share action of p leaks the classified data of the user's geolocation to facebook. Here we refer to the property share as a *Sink*. One thing to note is that if Line 5 is moved to after Line 6 that sets the value of s to p, then p is not considered as classified. This is because the type of s, **String**, is a value type, and thus the update of s does not affect p.

Now let's look at another example shown in Figure 4. At Lines 5, the message msg is added to the message collection msgs. The message collection msg keeps the references to msg, which means that msg can be accessed from msgs at a later time. At Line 6, msg becomes classified, which causes msgs to be classified indirectly. At line 7, msg2, the i-th message in msgs, may contain the data of msg or other messages. Thus, msg2 should also be considered as classified. We refer to this type of information

```
1  action foo(msg : Message, msgs: MessageCollection, i: Number) : Nothing {
2      pic := senses → take camera picture;
3      pic → share('facebook','share a pic');
4      s := bar();
5      msgs → add(msg);
6      msg → set message(s);
7      msg2 := msgs → at(i);
8      msg2 → share('facebook');
9      y := false;
10     if s → contains('Seattle') then {
11         y := true;
12     }
13 }
14
15 action bar() returns r : String{
16     l := senses → current location;
17     r := locations → describe location(l);
18 }
```

Figure 4: Example implicit and reference-type classified information flows.

flow as reference-type flow, since it occurs through objects such as message collections that contain references to other objects such as messages.

Another type of information flow that can potentially leak private information is implicit flow [9, 10]. Implicit flow arises from control structures such as *if* statements where the condition depends on classified information. The statements in the branches of the conditional statement can leak the outcome of the condition which allows later code to determine the classified information indirectly. Consider the example of implicit flow shown in Figure 4. The classified local s is used at the **if** statement at Line 10. By observing the values of y, users can guess whether the geolocation data stored in s contains the substring Seattle. Thus, to prevent such kind of information leak, we need to consider y as classified.

3 Application Capability Identification

The application capabilities tell users what kinds of mobile-device resources (such as sensors and wireless network) an application uses, which is useful information for users to decide whether to install the application. For example, if a game like minesweeper [4] has the capabilities of accessing your contacts and web, then it probably may leak your contacts to the web. These resources can be classified as sources (such as camera or geolocation) and sinks (such as web or facebook sharing). To use these resources, application developers need to use the APIs provided by the device-specific development environment, also called software development kit (SDK). Table 1 shows the kinds of sources and the sinks provided by the TouchDevelop APIs.

Some mobile-device platforms such as iOS prompt the user with a dialog when an

	Capability	Description
Source	Camera	Takes a picture through the camera.
	GeoLocation	Gets the geo location, possibly using GPS.
	Picture	Accesses the picture libraries.
	Music	Accesses the music.
	Microphone	Records the microphone.
	Contacts	Chooses emails or phone numbers.
Sink	Contacts	Saves an email or phone number to the phone.
	Media	Saves pictures to the phone.
	Sharing	Share information through social services, email or short messages.
	Web	Accesses the web, downloading or uploading data.

Table 1: Capabilities (sources and sinks) provided by the TouchDevelop APIs

application tries to access certain resources for the first time. However, iOS only inform users of geolocation information obtained from the GPS, ignoring other resources such as phone contacts or web accesses. Other mobile-device platforms such as Android use an install-time manifest to show all the resources that the application needs to access. However, as we mentioned in Section 1, developers have to specify these resources and they tend to claim more than needed [26]. These two ways of reporting application capabilities may fail to provide users the accurate and complete information of what resources are actually used by application. Users may have to trust the application reviewers or developers for the “claimed information” and make uninformed decisions at application install time.

Automated Application Capability Identification: To assist users in making informed decision based on the accurate and complete information of what resources are access by applications, our approach provides a static analysis that scans through the application script to automatically identify application capabilities. We have annotated all TouchDevelop APIs with source and sink information. We use a fixpoint algorithm to compute the capabilities used by each action of a script. For each action in a script, our approach parses the action into an abstract syntax tree (AST), and automatically scans each statement node in the AST to identify what sources and sinks are used. If a statement in an action a_1 is an action call to another action a_2, our approach unions the sources and sinks of a_2 to a_1. A fixpoint is reached if the computed sources and sinks for each action do not change. Since application developers can only use the APIs provided by the device-specific SDK for accessing mobile-device resources, our analysis results are guaranteed to be accurate and complete.

4 Static Information Flow Analysis

In this section, we present the overview of our static information flow analysis and then describe details of how our approach statically computes information flows using summaries computed by static symbolic execution [17].

4.1 Overview

Our approach statically computes information flows using summaries of basic blocks and actions. To compute the summaries, our approach simulates program executions, statement by statement, using static symbolic execution [17]. Our approach maintains the abstract state of the script and updates the state according to the simulated execution of a statement. The state maps local variables to sets of sources. In addition it maps a single mutable location for each type to sources. Finally, the state maps *sinks* to sources flowing to that sink. Sinks can be thought of as additional mutable locations that accumulate what flows into them. Information flow from a source s_1 to a sink s_2 arises whenever source s_1 appears in the abstract state of sink s_2. The sources in our maps are represented as a set of value elements consisting of constant sources and input parameter names. Input parameter names are used to represent symbolic information that allows us to determine where parameters flow.

In order to handle implicit flow arising from control flow statements that branch on classified information, we use an additional special local variable named pc. The pc variable is assigned (augmented) with source information at conditionals at the entry both branches. At each basic block, the pc is defined by the value of pc at the immediate dominator block instead of all predecessor blocks as is the case for normal locals.

Our approach uses a fix-point algorithm to iteratively compute the summaries of basic blocks in an action and then uses the summaries of basic blocks to compute summaries of actions. By instantiating the summaries with concrete values for symbolic parameter names and global variables, we can compute summaries of actions calling other summaries as well as for recursive actions.

4.2 Summaries of Basic Blocks and Actions

We separate the state into three parts: 1) local variable information, 2) pc information for implicit flow, and 3) mutable state information. The first two are program point specific, the mutable state is not. The mutable state consists of one classification per kind, and a set of edges between kinds representing possible references from the mutable state of objects of one kind to objects of another kind.

$$\begin{aligned}
\mathit{Atom} &::= \mathit{Sources}(i) \mid \mathit{Parameter}(i) \mid PC_{in} \\
\mathit{Classification} &::= \mathit{Set\ of\ Atom} \\
\mathit{LocalMap} &::= \mathit{Block} \rightarrow \mathit{Local} \rightarrow \mathit{Classification} \\
\mathit{SinkMap} &::= \mathit{Block} \rightarrow \mathit{Sink}(i) \rightarrow \mathit{Classification} \\
\mathit{PCMap} &::= \mathit{Block} \rightarrow \mathit{Classification} \\
\mathit{MutableState} &::= \mathit{Kinds}(i) \rightarrow \mathit{Classification} \\
\mathit{References} &::= \mathit{Set\ of}(\mathit{Kinds}(i) \times \mathit{Kinds}(i))
\end{aligned}$$

The fixpoint computation then computes the following data structures:

$$\begin{aligned} L_{pre}, L_{post} &: \mathit{LocalMap} \\ PC_{pre}, PC_{post} &: \mathit{PCMap} \\ S_{pre}, S_{post} &: \mathit{SinkMap} \\ M_{pre}, M_{post} &: \mathit{Block} \to \mathit{MutableState} \\ R_{pre}, R_{post} &: \mathit{Block} \to \mathit{References} \end{aligned}$$

L_{pre} contains the local information on entry to a particular block, whereas L_{post} contains the corresponding information at exit of the block, and similarly for PC_{pre} and PC_{post}. The sink maps S_{pre} and S_{post} contain the classification of the predefined sinks on entry and exit of blocks. M_{pre} and M_{post} contain the mutable state classification and R_{pre} and R_{post} contain the reference links between mutable states.

4.2.1 Block Summary

We initialize L_{pre} for entry blocks of actions to map each parameter local i to the singleton $\{Parameter(i)\}$ and to the empty set for all other locals. Similarly, we initialize PC_{pre} for entry blocks to the singleton $\{PC_{in}\}$ which allows computing symbolic summaries of actions that can be applied in contexts where the PC is classified differently. The sink map S_{pre} for the entry block is empty. These maps will not change during the global fix point of the analysis.

The information for R_{pre} and M_{pre} for the entry block keep track under which assumptions the action has been analyzed. It is initially empty, but may grow as the action is invoked in a context with larger M or R, causing the blocks of the action to be re-analyzed.

For non-entry blocks, the starting state is defined as follows:

$$\begin{aligned} L_{pre}(b) &= \bigsqcup_{b' inpred(b)} L_{post}(b') \\ S_{pre}(b) &= \bigsqcup_{b' inpred(b)} S_{post}(b') \\ M_{pre}(b) &= \bigsqcup_{b' inpred(b)} M_{post}(b') \\ R_{pre}(b) &= \bigcup_{b' inpred(b)} R_{post}(b') \\ PC_{pre}(b) &= PC_{post}(dom(b)) \end{aligned}$$

The locals on entry to a block are simply the union of the post local state of all predecessor blocks, where union is defined point-wise on the map (similarly for the sinks, mutable state, and reference links). For the PC classification is obtained by the post PC classification of the immediate dominator of block b.

4.2.2 Action Summary

We assume each action has a single exit block. The summary of an action is simply the post state of the exit block of the action. For each action, we keep track of the initial M and R under which it was analyzed in the information for its entry block. If we see a call to the action with a larger M or R, we update that information for the entry block and propagate the changes through the blocks of the action.

An example summary of the action foo in Figure 4 is:

$$
\begin{aligned}
\mathit{State} = \{ & \\
& L = \{s \rightarrow \mathit{GeoLocation}, \mathit{pic} \rightarrow \mathit{Camera}, y \rightarrow \mathit{GeoLocation}, \\
& \mathit{msg} \rightarrow \mathit{GeoLocation}, \mathit{msg2} \rightarrow \mathit{GeoLocation}\}, \\
& S = \{\mathit{Sinks}(\mathit{1}) \rightarrow \mathit{Camera}\}, \\
& PC = \{\}, \\
& M = \{\mathit{Picture} \rightarrow \mathit{Camera}, \mathit{Message} \rightarrow \mathit{GeoLocation}\} \\
& R = \{< \mathit{MessageCollection}, \mathit{Message} >\} \\
& \}
\end{aligned}
$$

Here the state of locals L shows that the local s contains the geolocation data, pic contains the camera data, y contains geolocation data due to the implicit flow from s to y, and the local msg gets geolocation data from s at Line 5. The state of mutable locations M shows that the mutable state of Picture contains the camera data and the mutable state of **Message** contains the geolocation data. The state of references R contains a pair showing that **MessageCollection** is linked to **Message**. Due to this link, msg2 reads the mutable data of msgs and is considered to contain the geolocation data. The state of sinks S shows that the sharing sink, pic →share, contains camera data, which come from the local pic. The states of pc is empty, since the pc does not carry the camera data after the **if**−**then**−**else** block.

4.2.3 Script Analysis

Since users can invoke actions within a script manually and one invocation can communicate information to another invocation via globals and mutable state, we have to analyze scripts as the following synthetic code:

```
while (true) {
  if (∗) {
    ..,g_{a_11}..g_{a_1m_1} := a_1(.., g_{a_11}..g_{a_1m_1});
  }
  if (∗) {
    ..,g_{a_21}..g_{a_2m_2} := a_1(.., g_{a_21}..g_{a_2m_2});
  }
  ..
}
```

The loop simulates the repeated invocations performed by a user of an arbitrary action, where the state is maintained by the global variables passed from one action to another.

The analysis analyzes this loop starting with empty classifications for all values and an empty reference set. A fixpoint is reached if the state does not change from one iteration to the next.

Due to this loop, the final fixpoint will have equal classifications for all M and R at each program point. Thus, an implementation of the analysis can simplify the handling of M and R.

4.3 Classified Information Propagation

In this section, we describe how our approach propagates classified information in scripts.

4.3.1 Property Annotations

We assume that all primitive properties are annotated with a set **ReadsMutable** consisting of the parameter indices of parameters whose mutable state is read by the property. Similarly, there is a set **WritesMutable** consisting of the indices of parameters whose mutable state is written by the property. Additionally, we use index 0 in the **WritesMutable** set to indicate whether the mutable state of the result depends on the classification of all the inputs to the property. By default, we assume that all immutable parts of all parameters are read by a property and that all read parts flow into the result's immutable part. Additionally, the set **EmbedsLinks** contains the set of edges between kinds that are potentially established by the property as references from the mutable state of one value to another.

A set of source indices **Sources** indicates which predefined sources flow into the result value through this property invocation. Finally, a set of sink indices **Sinks** indicates to which predefined sinks the classification of the parameters flow.

This information is used during the statement-based propagation to compute the propagation effected by a property invocation.

4.3.2 Statement-Based Propagation

The following rules show the propagation of the state for each kind of instruction. We assume L, PC, M and R are the initial states, and L', PC', M' and R' are the post states.

Case $x := y$

$$\begin{aligned} L' &= L[x \mapsto L(y) \cup PC] \\ PC' &= PC \\ M' &= M \\ R' &= R \\ S' &= S \end{aligned}$$

Note how the PC classification flows into the new classification of x. This is needed to keep track of implicit flow.

Case $r := p(x_1..x_n)$ First we compute the input classification, which consists of the classification of all input parameters, the classification of all kinds for which there is a parameter annotated with **ReadsMutable**.

$$\mathit{Common} = \mathit{PC} \cup \textbf{Sources}_p \cup \bigcup_i L(x_i) \cup \bigcup_{j \in \textbf{ReadsMutable}_p} \mathit{Cl}(M, R, \mathit{kind}(x_j))$$

The helper function $\mathit{Cl}(M, R, i)$ computes the union of the classification of all kinds j reachable from i via edges in R. Note that $\mathit{Reach}(R, i, i)$ is true for all R.

$$\mathit{Cl}(M, R, i) = \{M(j) \mid \mathit{Reach}(R, i, j)\}$$

With this information, we can now update the result and the mutable state.

$$\begin{aligned}
L' &= L[r \mapsto \mathit{Common}] \\
\mathit{PC}' &= \mathit{PC} \\
M'(i) &= \begin{cases} M(i) \cup \mathit{Common} & \text{if } \exists j \in \textbf{WritesMutable}_p \text{ and } \mathit{Reach}(R, \mathit{kind}(x_j), i) \\ M(i) & \text{otherwise} \end{cases} \\
R' &= R \cup \textbf{EmbedsLinks}_p \\
S'(i) &= \begin{cases} S(i) \cup \mathit{Common} & \text{if } i \in \textbf{Sinks}_p \\ S(i) & \text{otherwise} \end{cases}
\end{aligned}$$

Case $\mathit{assume}(x)$ **or** $\mathit{assume}(\mathit{not}\ x)$

$$\begin{aligned}
L' &= L \\
\mathit{PC}' &= \mathit{PC} \cup L(x) \\
M' &= M \\
R' &= R \\
S' &= S
\end{aligned}$$

Assume statements cause the PC classification to be augmented with the classification of the condition.

Case $r_1..r_n = a(x_1..x_m)$ First, we update $M_{pre}(\mathit{entry}_a)$ to $M \sqcup M_{pre}(\mathit{entry}_a)$ and $R_{pre}(\mathit{entry}_a)$ to $R \sqcup R_{pre}(\mathit{entry}_a)$. If necessary, propagate changes through blocks of a. We use the state at the exit block of a as the summary of a to be applied at the current invocation. Since the summary contains some symbolic information for parameter classification and pc classification, we first instantiate the exit block information with the invocation site information. Let σ be the substitution

$$\sigma = [\mathit{PC}_{in} \mapsto \mathit{PC}, \mathit{Parameter}(i) \mapsto L(x_i)]$$

Now we compute instantiated versions of the exit block summaries:

$$
\begin{aligned}
L_s &= \sigma(L_{post}(exit_a)) \\
M_s &= \sigma(M_{post}(exit_a)) \\
R_s &= \sigma(R_{post}(exit_a)) \\
S_s &= \sigma(S_{post}(exit_a))
\end{aligned}
$$

Note that no PC information flows out of the action. Let $r'_1..r'_n$ be the result locals in action a. The final states after the invocation of action a is then:

$$
\begin{aligned}
L' &= L[r_i \mapsto L_s(r'_i)] \\
PC' &= PC \\
M' &= M \sqcup M_s \\
R' &= R \cup R_s \\
S' &= S \sqcup S_s
\end{aligned}
$$

5 UnTampered- and Tampered-Classified Information

The source to sink information flow we compute so far may not be enough to make good policy decisions about which scripts are good and which scripts are bad. For example, a script taking a picture with the camera and then posting it to facebook may be a reasonable script, especially since posting to facebook will prompt the user and display the text and picture that will be posted. The user thus has a way to vet the information being posted to a degree.

However, a malicious script could try to encode the user's phone number into the color intensity of some pixels in the posted picture. From an information flow perspective, we would simply see that sources camera and contact flow to share. The users looking at the picture being posted will likely not notice changed pixels containing the hidden phone number.

Can we distinguish somehow between these two cases? Our attempt to do so is based on the following assumption: for sinks that prompt the user to review the information (e.g., emails, sms, phone calls, facebook posts), we want to distinguish if the information being posted is recognizable by the user as containing sensitive information or not. In the case where pixels in the picture taken by the camera are modified based on classified contact information, we want to consider the information in the picture as tampered and thus apply a harsher policy than if the information is not tampered with.

In order to track tampering, we introduce an operator *Tamper* that can be applied to the existing sources.

$$
Atom ::= Sources(i) \mid Parameter(i) \mid PC_{in} \mid Tamper(Atom)
$$

Note that the set of atoms is not unbounded, as this is not a free algebra. Indeed, $\mathit{Tamper}(\mathit{Tamper})(s) = \mathit{Tamper}(s)$ for all s.

Additionally, we annotate all properties p with a single bit $\textbf{Tampers}_p$ indicating whether any input classifications are transformed into tampered output classifications for the result and writes to the mutable store.

The rule for handling the flow at property invocations then needs to be modified insofar as the classification *Common* now becomes:

$$\mathit{InFlow} = PC \cup \bigcup_i L(x_i) \cup \bigcup_{j \in \textbf{ReadsMutable}_p} Cl(M, R, \mathit{kind}(x_j))$$

$$\mathit{Common} = \textbf{Sources}_p \cup \begin{cases} \mathit{InFlow} & \text{if } \neg\textbf{Tampers}_p \\ \mathit{Tamper}(\mathit{InFlow}) & \text{if } \textbf{Tampers}_p \end{cases}$$

Applying *Tamper* to an entire classification, just means applying the operator pointwise to the set elements.

6 Transparent Privacy Control

By applying the static analysis, we compute information flows on a per action and per script basis and show summaries of which sources flow to which sinks in each action and in the script as a whole. As an example, Figure 1 shows the summary of the script named *location and maps*, which can send a text message containing the user's current location or take a picture with the user's current location embedded in it and save the picture into the media storage library of the mobile device. This flow summary transparently shows the information flows of the application: by looking at the information flows at install time, users can understand what private information the application uses and where this private information may escape.

Once a user install application scripts, they can run the scripts on their devices and grant the scripts access to real information if desired. The first time a user runs a script, our approach allows the user to use *anonymized* data for each source that flows into a sink or *abort* execution at the access point. In this way, our approach prevents leaking of a user's private information while preserving the fun and utility of applications to a reasonable extent. To minimize a user's effort in granting accesses to sources, we further define a policy that classifies flows into safe and unsafe flows, which are used to provide different default settings.

Classification of Safe and Unsafe Flows: Our policy is based on the assumption described in Section : we consider a flow as a safe flow if it is an untampered flow to a *vetted sink*. A vetted sink is a sink that results in an explicit dialog at runtime, presenting the particular information flowing to the sink to the user and asking the user to confirm the flow. For example, a post to facebook would prompt the user to review the information before the actual sharing happens. Since the data is untampered, the user can make an informed choice of whether or not to share the data. We may evolve the policy of what constitutes a safe flow based on user feedback, and we will update the policy when more sources and sinks are added into the system. Our approach considers as unsafe flows the other flows, including untampered flows to unvetted sinks and all tampered flows.

Granting Accesses: When running the script for the first time, the user is presented with all sources flowing to unsafe sinks along with a radio button group for each source that allows the user to choose among *anonymized* information, *real* information, or *abort* execution [3] for that private information, shown in Figure 2. Anonymized information means that the runtime provides the script with anonymized information (a fixed picture or a fixed geolocation etc.), Real information means the script gets access to the real information on the users' devices, and abort execution means that the runtime stops the execution at the access point. By using the anonymized data, the user can safely experiment with an application to determine if it does something useful prior to even considering whether to allow access to real information. Additionally, the abort execution option provides a stack trace at the access point, which is valuable for those users who are able to inspect the source code in order to diagnose the flows.

Default Settings of Accesses: To minimize users' efforts in granting accesses, our approach provides default settings: for sources that appear only in safe flows, the default setting is to use real information; for other sources, the default setting is to use anonymized information. This default setting provides a reasonable way for users to simply confirm the accesses while protecting their private information. The anonymized/real/abort settings for each kind of private information can be changed at will by users in the settings for that script. Users can also choose the option of *all anonymized* on the top or the *all real* option on the bottom (so users have to scroll down before they can choose this option).

7 Evaluation Plan

This section presents the evaluation plan of our approach. We integrated our static information flow analysis as part of the TouchDevelop [6] website, where users can search, comment and install published scripts. Our analysis analyzes every submitted script to compute capabilities and information flows, which are then shown to users in the script installation page. We built a prototype of transparent privacy control into the TouchDevelop App [5]. Our prototype prompts users the first time a installed script runs and asks users to grant accesses to private information. We plan to conduct three evaluations to evaluate the effectiveness of computing information flows, the performance of information flow analysis, and the user experiences of our privacy control and default settings.

7.1 Information Flow Study

TouchStudio [25], the previous version of TouchDevelop, was one of the 100 most downloaded apps among over 13,000 apps on 4/14/2011. With the exciting script shar-

[3] The *abort* behavior is not implemented in the first released version of TouchDevelop.

ing and many other improvements, we expect TouchDevelop to receive similar or better download records and we would receive many submitted scripts as our study subjects. We plan to select part of these submitted scripts to study whether our static analysis can effectively identify information flows.

7.2 Performance Evaluation

To analyze all the scripts submitted by users, the static analysis must have acceptable performance even if it is running on the cloud server. We plan to collect a certain number of submitted scripts of different sizes and study the analysis time on the cloud server. This performance study can provide us insights on whether some optimization techniques are required or more precise static analysis can be pursued.

7.3 Usability Study

To study the user experiences of our privacy control and to evaluate our default settings, we plan to perform a user survey with a portion of TouchDevelop users. This user study can provide us feedbacks on whether our privacy control overwhelms users. We plan to build a survey dialog into the TouchDevelop App and randomly select a number of users to participate our survey. We expect to randomly choose at least 300 users to participate our survey. To fit the questions into the touch screen (i.e., no need to scroll) and not introduce too much burden on users for the survey, we plan to ask three questions about our privacy control and default settings:

1. Do you feel your privacy is respected and protected?
   Yes/ No/ Don't Care

2. Which option did you like?
   All anonymized or All real/ Individual Settings/ Don't Care

3. Do you like our default settings?
   Yes/ No/ Don't Care

Additionally, we plan to study the user interactions with the TouchDevelop app as a substitute measure of user experience. We plan to instrument the buttons clicked by users and use this data to infer the user experiences of our privacy control:

1. We collect the data of number of clicks on "capabilities" and "information flow" buttons. This data can show whether information of capabilities and information flows is useful for users at install time.

2. We collect the data of number of clicks on "all anonymized/real" and "individual setting" buttons. This data can show whether users prefer to using "all anonymized/real" setting or using "individual setting" for each source.

3. We use the length of user interaction with TouchDevelop app to infer the user experience of our privacy control. For example, we can learn from the user interactions that how many users stop using TouchDevelop app after some time,

such as a week. When users uninstall the TouchDevelop App, we will ask users to fill in the reason why they do not like the TouchDevelop app and study how many reasons are related to our privacy control.

8 Related Work

In this section, we compare our work with other approaches that identify application capabilities, track information flows, grant accesses to private information, and automatically validates mobile apps.

User-Aware Application Capabilities: Mobile-device platforms like Android [1] and social-network platforms like Facebook [2] use manifests to show application capabilities and request permissions at install time. As we discussed in Section 3, the capabilities shown in the manifests are claimed by developers and are not accurate or complete, since developers tend to claim more capabilities than needed, violating the principle of least privilege [24]. Other mobile-device platforms like iOS [3] and research approaches like TaintDroid [11] report application capabilities the first time applications try to access a resource. Although the reported capabilities are accurate, these approaches only present part of the application capabilities. For example, iOS only shows the capability of geolocation to users; TaintDroid reports only the capabilities accessed by the executed code and cannot detect the capabilities in the unexecuted code. Our approach provides annotations on all the APIs used to access resources on devices and applies static analysis to compute application capabilities, which is guaranteed to be accurate and complete.

Wetherall et al. propose a concept called *privacy revelation* [27], which requires that (1) users must be aware of the spread of personal information based on user-relevant context; (2) users should be able to give feedback before information exposure; (3) users can learn from other users' experiences. Part of our approach can be considered as one instance of their concept, since our approach reveals information flows to users and requests users to grant access to private information. However, our approach adapts static information flows analysis to expose information at both the install time and the first time users run the applications, while their developing systems are all based on dynamic analysis, which cannot provide information before users even installs an application. Moreover, our anonymize/real/abort setting encourages users to try out applications with safe default settings, while their approach encourages sharing of privacy revelations.

Information Flow Analysis: Xie and Aiken [28] present an approach that statistically detect security vulnerabilities in scripting languages. Their approach statically computes summaries of blocks and procedures of PHP and detects security vulnerabilities at the block level, intraprocedural level, and interprocedural level. Since their approach is tuned to focus on detecting SQL injection and cross site scripting (XSS) vulnerabilities [16], their approach cannot be directly applied to compute general information flows. For example, their approach does not handle reference-type flows shown in Figure 4, and would lose track of flows after built-in procedure calls (e.g., senses → take camera picture) that cannot be analyzed by their approach. The hybrid analysis approach proposed by Chandra et al. [8] primarily tracks the information flow at runtime, but uses static analysis to flow security labels to variables defined or updated in non-executed branches. Although our approach is similar to these two approaches in computing information flows using static analysis, our approach further uses mutable locations to simplify analysis of reference-type flows and tracks untampered- and tampered-classified information for classifying safe and unsafe flows.

Language-based information flow [23] allows developers to annotate variables with security attributes. These attributes can be used by compilers to enforce information flow controls. For example, Slam [15] shows that information flow labels can be applied to a simple language with reference types and Jif [20, 21] extends Java language with statically-checked information flow annotation. Laminar [22] allows developers to specify security regions and provide information flow controls on both language and JVM/OS levels. Although these language-extending approaches are effectively in guaranteeing information flow controls, they impose additional burdens on developers when writing applications, which is undesirable for writing scripts on mobile devices in the context of TouchDevelop.

Dynamic taint analysis [11,30] has been applied to track information flows on both mobile platforms like Android and desktop platform like Windows. These approaches track tainted data during runtime and monitor the behaviors of applications for identifying information leaks. These approaches can precisely identify information leaks if there is any tainted data trying to escape, providing accurate runtime information about the leaks for analysis. However, to reduce runtime overhead, these approaches usually ignore implicit flows raised by control structures. Moreover, to detect potential information leaks for all submitted applications, dynamically executing these applications to detect potential information leaks requires too much overheads, and it is impractical to execute all possible paths. These limitations make these approaches inappropriate on computing information flows for all submitted applications.

Access Granting: Mobile-device platforms like Android [1] and social-network platforms like Facebook [2] use manifests to request permissions at install time. Once permissions are given by users, the permissions cannot be changed. Since only the information of the "claimed capabilities" is given to users at install time, users cannot make informed decisions on granting accesses and usually grant all the accesses even if applications ask for excessive permissions [26]. To assist users in making informed decisions on granting access, our approach presents information flows to describe how applications may do with users' private information. Our anonymized/real/abort access granting also provides a way for users to try out applications before using private information, and these accesses can be changed at will.

iOS and Windows User Account Control [18] prompts a dialog to request permissions from users when applications try to access a resource or make security/privacy-related system-level changes. Similarly, these prompt approaches fail to provide information about the usage of the resources for users to make decisions and users become habituated to just allow the accesses [19, 29]. Besides providing application capabilities and information flows, our approach provides sensible default settings of accesses to resources in order to minimize users' efforts in granting accesses. In this way, users can safely experience applications even if they just accept the default settings.

Zhu et al. propose an approach that uses application-level dynamic taint analysis to track the movement of sensitive user data as it flows through applications [30]. When tainted data is found before the system call is about to enter, their approach allows users to choose among logging the action, blocking the system call, or randomize the tainted data. Our anonymized/real/abort setting is inspired by their approach and we extend their access granting settings by using static information flow analysis. Based on the information flows analysis extended to track tampered/untampered classified information, our approach defines policies to classify safe and unsafe flows and assign different default accesses, requiring less efforts from users. Thus, given a flow describing that a picture taken from the camera is shared directly via facebook (a vetted sink), our approach would consider it as benign and uses real information as the default setting, while their approach can only identify it as an unsafe flow and require input from users.

Automated Security Validation of Mobile Apps Gilbert et al. present a vision of making mobile apps more secure via automated validation [14]. They propose using commodity cloud infrastructure to emulate smartphones and run the submitted apps to dynamically track information flows and actions. Based on the information flow and action tracking, they propose to automatically detect malicious behavior and misuse of sensitive data via further analysis of dependency graph [13] or natural language processing. Although our information flow analysis is similar to their runtime tracking, our approach further tracks whether information is tampered before the information exposure and lets users use anonymized information to try out applications instead of defining policies to detect malicious behaviors.

9 Conclusion

Central application downloading service provided by modern mobile-device platforms allows users to download third-party applications, adding more functionalities into mobile devices. However, these applications may leak private information without notifying users. Existing approaches provide inaccurate or incomplete information about resources used by applications, failing to assist users in making informed decisions on whether to install applications and controlling their privacy. We present a transparent privacy control approach, which adapts static analysis to compute information flows and allows users to control their privacy by using anonymized information or abort execution instead of using private information. We built a prototype of our approach into TouchDevelop, an application-creation environment that let users to write applications using the touchscreen and install written applications published by other users. We plan to conduct evaluations to study the effectiveness of static information flow analysis and usability of our privacy control.

Acknowledgment

We would like to thank all researchers and developers at Microsoft Research who helped to shape our approach in countless discussions.

References

[1] Android. `http://www.android.com/`.

[2] Facebook. `http://www.facebook.com/`.

[3] iOS. `http://www.apple.com/ios/`.

[4] Minesweeper. http://en.wikipedia.org/wiki/Minesweeper_(computer_game).

[5] TouchDevelop. `http://research.microsoft.com/TouchDevelop`.

[6] TouchDevelop Website. `http://www.touchdevelop.com/`.

[7] Windows Phone. `http://www.microsoft.com/windowsphone/`.

[8] Deepak Chandra and Michael Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Annual Computer Security Applications Conference (ACSAC)*, pages 463–475, 2007.

[9] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, pages 236–243, 1976.

[10] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of The ACM*, pages 504–513, 1977.

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI*, pages 1–6, 2010.

[12] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The Effectiveness of Application Permissions. In *Proc. WebApps*, pages 7–7.

[13] Jeanne Ferrante and Karl J. Ottenstein. The Program Dependence Graph And Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[14] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps At App Markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, 2011.

[15] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy And Integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[16] G. Hoglund and G. Mcgraw. *Exploiting Software: How To Break Code*. Pearson Education, 2004.

[17] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[18] MICROSOFT. What is User Account Control?, 2011. http://windows.microsoft.com/en-US/windows-vista/What-is-User-Account -Control.

[19] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows Users Follow The Principle of Least Privilege?: Investigating User Account Control Practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, 2010.

[20] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[21] Andrew C. Myers and Barbara Liskov. Protecting Privacy using The Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, pages 410–442, 2000.

[22] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. Mckinley, and Emmett Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–74, 2009.

[23] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 2002.

[24] J H Saltzer and M D Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, pages 1278–1308, 1975.

[25] Nikolai Tillmann, Michal Moskal, and Jonathan de Halleux. TouchStudio - Programming Cloud-Connected Mobile Devices via Touchscreen. *Microsoft Technical Report MSR-TR-2011-49*, 2011.

[26] T. Vidas, N. Christin, and L. Cranor. Curbing Android Permission Creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.

[27] D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang. Privacy Revelations for Web And Mobile Apps. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[28] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, 2006.

[29] Ka-Ping Yee. Aligning Security and Usability. *IEEE Security and Privacy*, 2:48–55, 2004.

[30] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, pages 142–154, 2011.