User-Aware Privacy Control via Extended Static-Information-Flow Analysis

Xusheng Xiao[1]* Nikolai Tillmann[2] Manuel Fahndrich[2]
Jonathan de Halleux[2] Michal Moskal[2]

[1]Dept. of Computer Science, North Carolina State University, Raleigh, NC, USA
[2]Microsoft Research, One Microsoft Way, Redmond, WA, USA
[1]xxiao2@ncsu.edu, [2]{nikolait, maf, jhalleux, micmo}@microsoft.com

ABSTRACT

Applications in mobile-marketplaces may leak private user information without notification. Existing mobile platforms provide little information on how applications use private user data, making it difficult for experts to validate applications and for users to grant applications access to their private data. We propose a user-aware privacy control approach, which reveals how private information is used inside applications. We compute static information flows and classify them as safe/unsafe based on a tamper analysis that tracks whether private data is obscured before escaping through output channels. This flow information enables platforms to provide default settings that expose private data only for safe flows, thereby preserving privacy and minimizing decisions required from users. We built our approach into TouchDevelop, an application-creation environment that allows users to write scripts on mobile devices and install scripts published by other users. We evaluate our approach by studying 546 scripts published by 194 users.

Categories and Subject Descriptors

D.2.4 [**Software Verification**]: Validation; K.4 [**Computers and Society**]: Privacy

General Terms

Human Factors, Security, Verification

Keywords

Mobile Application, Privacy Control, Information Flow Analysis

1. INTRODUCTION

Modern mobile-device platforms like iOS, Android, and Windows Phone provide a central place, called app stores or marketplaces, for finding and downloading third-party applications. A common problem faced by these mobile-device platforms is that the published applications in the marketplace may leak private user information through output channels. Many of these applications access mobile-device resources, such as pictures and GPS that may contain and expose private information, and share them using remote cloud services or web services without notifying users [1].

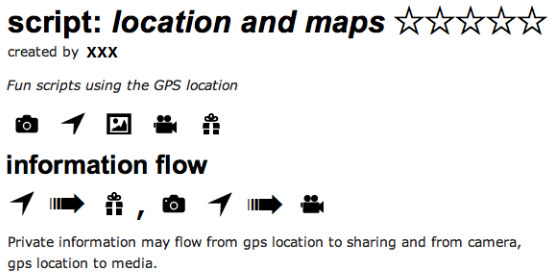

script: ***location and maps*** ☆☆☆☆☆
created by **XXX**
Fun scripts using the GPS location
information flow
Private information may flow from gps location to sharing and from camera, gps location to media.

Figure 1: Information flow view of a sample script

To mitigate these problems, privacy control mechanisms employed by mobile-device platforms include two major parts: (1) manual app validation by experts: experts employed by an app store manually exercise the functionality provided by an app and observe its behaviors for validation; (2) access-control granting by users: app stores ask for permissions before users can install applications (Android and Windows Phone), or an app requests permissions before it can access users' private information (iOS). The manual validation process is costly and delays publishing of apps. It is also incomplete, since it cannot examine every execution path to detect violations of privacy policies [2]. Access-control granting provides information about *what* private information these applications may access, rather than *how* these applications use private information, causing users to make uninformed decisions on how to control their privacy. These privacy control mechanism lead to a situation where users simply install applications without questioning the requested permissions, even if the applications may silently leak private information [1, 3, 4].

To improve the privacy control mechanism of these mobile platforms, we provide a user-aware privacy control approach that reduces efforts for app validation and access-granting by computing information flows and classifying information flows as safe/unsafe.

Our approach automatically computes information flows of private information via static analysis and visualizes the flows, as shown in Figure 1. We use the term *Source* to refer to an origin of private information and *Sink* to refer to a point where information may leak from an app. The example in Figure 1 shows that the app uses 5 capabilities (Camera, Location, Pictures, Media, and Sharing). Among these, the first 3 are sources, and the last two are sinks. Among the 6 possible flows (3 sources to 2 sinks), our analysis shows that the Location flows to the Sharing sink, and that Camera and Location flow to the Media sink.

*Worked on this project as a Microsoft Research intern.

Given the computed information flows, our approach employs the mechanism of user-driven access control [5]. When the application is executed for the first time, our approach allows users to choose among *real* information, *anonymized* information, or *abort* execution, as shown in Figure 2 (the *abort* option is not yet implemented in TouchDevelop). These settings provide flexible choices for users: (1) using anonymized information (e.g., a fixed picture or a fixed geolocation), users can experiment with applications before granting access to real information; (2) aborting an execution prevents unintended access to a resource and is helpful for diagnosis.

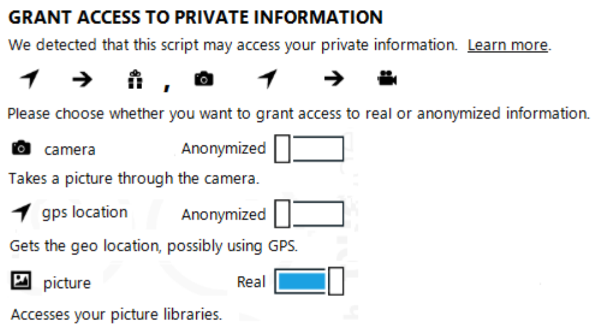
To assist experts and users in better understanding how apps handle private information and improve privacy control, our approach further classifies information flows based on a tamper analysis. We define a policy to classify information flows as safe or unsafe: an information flow is safe if only *untampered* private information flows to a *vetted* sink. A vetted sink is a sink that presents an explicit dialog requesting the user's permissions before the information being shown escapes. In Myer's terminology [6], this dialog corresponds to a declassify step and tampered data has low integrity. For example, in TouchDevelop [7], the sharing of a picture taken directly from the camera shows a dialog for users to review the picture before it leaks from the device. Such information flows do not leak private information without notifying users and should be safe. However, a malicious app could 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. Our analysis detects such obscure flow by observing whether the information is tampered with before reaching the sinks. Based on the safe/unsafe classification of flows, our policy is to use real information for sources only appearing in safe flows, and anonymized information for all other sources.

Our user-aware privacy control approach strives for a balance between security and user involvement. By employing user-driven access control, our approach ensures that apps gain permissions from users for private information accessed by apps. To avoid overwhelming the users with access granting—which may annoy users and cause users to blindly grant every permission—our approach does not ask users to grant access to private information accessed by an app but not flowing to sinks. Furthermore, our technique provides default settings that are safe to run a script without further user decisions, thereby reducing risk and user burden.

We built a prototype of our privacy control into TouchDevelop, a novel mobile platform that enables users to write apps directly using touch screens. In TouchDevelop, apps are written using a scripting language that is expressive enough to create applications or games, utilizing most features of mobile devices [8]. We call apps written in TouchDevelop "scripts". Users can publish their scripts in a "script bazaar", where other users can install and run them on their own devices. TouchDevelop is thus similar to other mobile-device platforms, except that we use no manual validation, only automatic information flow analysis. Our approach works well with the TouchDevelop platform for several reasons: (1) all code is made available through the script bazaar as source; (2) the expressiveness of the language enables apps to be created in fewer lines, allowing efficient static analysis on whole scripts; (3) the language does not allow reflection, eval, or native calls to platform APIs, making code analysis easier [9].

This paper makes the following contributions:

- We propose a user-aware privacy control approach, which reveals information flows and their classification to users to assist app validation and user-driven access control. Our approach is a first step towards improving privacy control on mobile devices via automatic analysis.


GRANT ACCESS TO PRIVATE INFORMATION

We detected that this script may access your private information. Learn more.

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 present an extended static analysis to compute information flows and check tamper information for classifying information flows as safe/unsafe flows.

- The automatic computation and classification of information flows and resuling safe default settings enable our script bazaar to operate *without any manual script validation*.

- We built a prototype of our privacy control into TouchDevelop, both for analyzing published scripts, and to present user privacy settings to the user based on our analysis and policy.

- We studied 546 scripts published by 194 users to evaluate the effectiveness and performance of our information flow analysis. The results show that among the 546 scripts, 172 use a private source, but only 78 scripts (14.29%) flow private information to a sink. Among these 78 scripts, our approach classifies 24 as safe, reducing the need to make access granting choices to a mere 10.1% (54) of all scripts. Alternatively, users need to grant access to only 63 sources (41.4%) among 152 sources appearing in scripts together with sinks.

2. TouchDevelop LANGUAGE

TouchDevelop allows users to create applications using an imperative and statically typed language [8]. 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 persisted and accessible across multiple script invocations.

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., $p \rightarrow$ share). For the sake of the simplicity, the language does not provide features that allow users to define new types or properties.

2.1 Classified Information Flow

In this section, we illustrate several examples to show how scripts written in TouchDevelop 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, variable loc becomes classified since it contains the geolocation information obtained via the GPS. Here, we refer to the

```
1  action foo() : Nothing {
2      var s := "unclassified";
3      var p := media → create picture();
4      var loc := senses → current location; // classified
5      s := loc → describe(); // classified
6      p  → draw text(s); // p's mutable state is classified
7      p  → share("facebook");
8  }
```

Figure 3: Example of classified information flow

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

Figure 4: Implicit and reference-type information flow

property senses → current location as a *Source* of geolocation information. 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 information 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 were moved to after line 6, then p would not be classified. The later update of s would not affect p.

Now let's look at another example shown in Figure 4. At line 5, the message msg is added to the message collection msgs. The message collection msgs keeps a reference 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 information of msg or other messages. Thus, msg2 should also be considered as classified. We refer to this type of information flow as reference-type flow, since it occurs through objects such as message collections that contain references to other objects.

Another type of information flow that can potentially leak private information is implicit flow [10,11]. Implicit flow arises from conditional 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 information stored in s contains the substring Seattle. Thus, to track implicit information flows, we need to consider y as classified.

3. CAPABILITY IDENTIFICATION

The application capabilities tell users what kinds of mobile-device resources (such as personally-sensitive information and wireless network) an application uses, which is useful information for users to decide whether to install the application. 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 sinks provided by the TouchDevelop APIs. Among these sinks, the sink *Sharing* prompts users with the sharing information, which makes it a vetted sink. For the other three kinds of sinks, only the sink *Web* is considered as an unvetted sink. The reason is that the pictures from the sink *Picture* and emails or phone numbers from the sink *Contacts* are all considered as sensitive private information, and if these kinds of private information would flow to *Web*, our approach would identify the flow as an unsafe flow.

Table 1: Capabilities provided by the TouchDevelop APIs

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

Automated Capability Identification. To provide the accurate and complete information of what resources are accessed by applications, our approach provides a static analysis that scans through the application script to automatically identify application capabilities. We have manually 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 a call to another action a_2, our approach adds 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 in TouchDevelop 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. INFORMATION FLOW ANALYSIS

In this section, we first present an overview of our static information flow analysis, and then follow it up with full technical details.

4.1 Overview

Our approach statically computes information flows using abstract interpretation [12]. 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 kind[1] to a set of 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

[1]Data types in TouchDevelop are called kinds.

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.

Implicit Flows. 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 of 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.

Inter-Procedural Analysis. Our approach uses a fix-point algorithm to iteratively compute the summaries of basic blocks in an action and then uses these summaries to compute summaries of actions. At call-sites, summaries are instantiated with concrete values for symbolic parameter names, thereby computing the effect of the call without re-analysis of the action. This approach also handles recursive actions.

Mutable and Immutable Values. We map the TouchDevelop concepts to a simpler model for information flow analysis. 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 parts, 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.

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 kind. Every value of that type shares that same mutable location in the analysis. All updates to the mutable part are weak updates, meaning they are accumulated.

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

Embedded References. Because values may have embedded references 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. 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. As a result, inside an action, accessing a global is no different than accessing a local variable. We will thus no longer explicitly talk about global variables henceforth.

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 treated as 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.

4.2 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.

4.3 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, but 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.

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

The fixpoint computation 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.

4.3.1 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:

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

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.

4.3.2 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. For example, the summary of action foo in Figure 4 is:

$$\begin{aligned}
State = \{ & \\
& L = \{s \rightarrow \{\mathbf{Location}\}, pic \rightarrow \{\mathbf{Camera}\}, \\
& \quad y \rightarrow \{\mathbf{Location}\}, msg \rightarrow \{\mathbf{Location}\}, \\
& \quad msg2 \rightarrow \{\mathbf{Location}\}\}, \\
& S = \{\mathbf{Sharing} \rightarrow \{\mathbf{Camera}\}\}, \\
& PC = \{\}, \\
& M = \{Picture \rightarrow \{\mathbf{Camera}\}, \\
& \quad Message \rightarrow \{\mathbf{Location}\}\} \\
& R = \{< MessageCollection, Message >\} \ \}
\end{aligned}$$

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 contains camera data. The set PC is empty, since the pc does not carry the camera data after the **if−then−else** block.

4.4 Classified Information Propagation

In this section, we describe how APIs are annotated and how information flow is tracked at the instruction level.

4.4.1 Property Annotations

We assume that every primitive property p is annotated with a set **ReadsMutable**$_p$ consisting of the parameter indices of parameters whose mutable state is read by p. Similarly, the set **Writes-Mutable**$_p$ consists of the indices of parameters whose mutable state is written by p. Additionally, we use index 0 in **WritesMutable**$_p$ to indicate whether the mutable state of the result depends on the classification of the inputs to property p. 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**$_p$ contains the set of edges between kinds representing possible references established by invoking property p.

A set **Sources**$_p$ indicates which predefined sources flow into the result value when invoking property p. Finally, **Sinks**$_p$ contains the set of sinks to which information flows on invoking p.

4.4.2 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**.

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

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

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

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

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

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

$$L' = L$$
$$PC' = PC \cup L(x)$$
$$M' = M$$
$$R' = R$$
$$S' = S$$

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

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

Now we compute instantiated versions of the exit block summaries:

$$L_s = \sigma(L_{post}(\mathit{exit}_a))$$
$$M_s = \sigma(M_{post}(\mathit{exit}_a))$$
$$R_s = \sigma(R_{post}(\mathit{exit}_a))$$
$$S_s = \sigma(S_{post}(\mathit{exit}_a))$$

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:

$$L' = L[r_i \mapsto L_s(r'_i)]$$
$$PC' = PC$$
$$M' = M \sqcup M_s$$
$$R' = R \cup R_s$$
$$S' = S \sqcup S_s$$

5. TAMPERED 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.

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 Contacts flow to Sharing. 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.

$$\begin{aligned} \mathit{Atom} ::= &\ \mathit{Sources}(i) \mid \mathit{Parameter}(i) \\ &\mid PC_{in} \mid \mathit{Tamper}(\mathit{Atom}) \end{aligned}$$

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:

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

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

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

6. USER-AWARE 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 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 to. To minimize the efforts of experts in validating applications and users in granting accesses to sources, we further define a policy that classifies flows into safe and unsafe flows.

Classification of Safe and Unsafe Flows. Our policy is based on the assumption described in Section 5: we consider a flow as a *safe flow* if it is an untampered flow to a *vetted sink*. Recall that a vetted sink results in an explicit dialog at runtime, presenting the particular information flowing to the sink and requesting permissions from the user before the information escapes from the mobile-device. For example, a post to facebook would prompt the

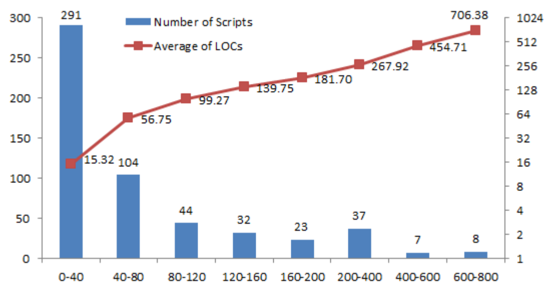


Figure 5: Sizes of 546 published scripts in TouchDevelop

user to review the information before the actual sharing happens. Our approach considers all other flows as unsafe, including untampered flows to unvetted sinks (Web) and all tampered flows. We may evolve the policy of what constitutes a safe flow based on user feedback, and update the policy when more sources and sinks are added into the system.

Granting Accesses. When running the script for the first time, the user is presented with all sources appearing in unsafe flows along with a radio button group for each source that allows the user to choose among *anonymized* or *real* information (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' device, and abort execution means that the runtime stops the execution at the access point. By using anonymized information, a 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.

Default Settings. To keep users safe and minimize efforts in granting access, our approach provides default settings. We guarantee that running a script with the default settings does not leak private information, except through vetted sinks where the user is presented untampered information to review. Sources appearing in no flows use real information and are not shown. For sources that appear only in safe flows, the default setting is to use real information; for other sources appearing in flows, the default setting is to use anonymized information.

7. EVALUATION

This section presents experiments we conducted to evaluate the effectiveness of our extended static information flow analysis. We chose TouchDevelop as a platform for our evaluations due to two major reasons: (1) **source code availability**: the source code of a script is made available as part of the publishing process; (2) **simplicity**: the expressiveness of the TouchDevelop languages enables applications to be created in much fewer lines, reducing the complexity of static analysis; the TouchDevelop language does not allow reflection or native calls to platform APIs, enabling complete annotation of the APIs with source, sink, and flow information; TouchDevelop only allows importing of external scripts through the script bazaar and does not allow generating code at runtime.

7.1 Subjects and Evaluation Setup

We integrated our static information flow analysis into the server part of the TouchDevelop environment. Every submitted script is analyzed automatically and the resulting flow information informs the privacy settings when users install scripts. To conduct the experiments, we collected 546 scripts (all publications prior to Oct

Table 2: Information flow summary of 546 published scripts

# Total	# Cap (242)			# Flow
	/w **Source**	/w **Sink**	/w **Both**	
546	172	159	89	78

6th, 2011) published by 194 TouchDevelop users, excluding scripts published by ourselves. Figure 5 shows the number of scripts in different ranges of lines of code (LOC) [2] and the average LOCs in these ranges. Among these scripts, 395 (72.34%) scripts have LOCs ranging from 0-80, and the scripts *Termini 3 Final*[3] and *Termini 3 Beta 1.4*[4]) have the maximum LOCs of 738. The major reason why these scripts are of relatively small size is that the expressiveness of the TouchDevelop language enables users to create applications using fewer lines of code than using traditional programming languages for mobile devices. For example, the script *Termini Include Edition 1.0.2*[5] published by the user Pouya Animation[6] creates a UNIX emulator (Terminal) for TouchDevelop in just 407 LOC.

7.2 Information Flow Evaluations

To show the effectiveness of our information flow analysis, we posed the following three research questions about the 546 subject scripts:

RQ1 : What is the advantage of using information flow from sources to sinks to classify scripts, as opposed to the mere presence of both sources and sink (capability usage)?

RQ2 : How many more scripts can we classify as safe using our tamper analysis, thus eliminating the need to ask users to grant access?

RQ3 : How many more sources can we classify as safe using our tamper analysis, further reducing the number of sources that require users' decisions?

7.2.1 RQ1: Information Flow Summary

To address RQ1, we compare the number of scripts that are classified as information-leaking using information flows with the number of scripts that are classified as information-leaking using capabilities. Table 2 shows the information flow summary of the published scripts. Column "# Total" shows the total number of scripts. Column "# Cap" shows the number of scripts that either have at least one source or one sink. Column "/w Source" shows the number of scripts that have at least one source. Column "/w Sink" shows the number of scripts that have at least one sink. Column "/w Both" shows the number of scripts that have both sources and sinks. Column "# Flow" shows the number of scripts that have computed information flows.

The results show that in 546 published scripts, 242 (44.32%) either have sources (access private information) or have sinks (can leak information from the script). To form an information flow, a script must have at least one source and one sink. As shown in Table 2, 457 (83.70%, #Total - #Both) scripts have either no sources or no sinks, which can be classified as non-information-leaking by either using information flow or capabilities usage. For the remaining 89 scripts that have both sources and sinks, our information flow

[2]Meta data and comment statements are excluded for LOC computation.
[3]http://touchdevelop.com/pycw
[4]http://touchdevelop.com/xwgl
[5]http://touchdevelop.com/hllw
[6]https://www.touchdevelop.com/ntqe

Table 3: Information flow vs. source-sink pairs

	Contacts	Media	Sharing	Web	*Any*
Camera	0 / 0	22 / 22	11 / 21	0 / 30	33 / 36
Contacts	1 / 3	0 / 11	30 / 39	0 / 19	30 / 41
Location	0 / 0	6 / 10	12 / 12	27 / 29	30 / 34
Microph.	0 / 0	0 / 2	1 / 1	0 / 3	1 / 6
Music	0 / 0	0 / 1	1 / 1	0 / 1	1 / 3
Picture	0 / 0	18 / 29	2 / 15	14 / 21	24 / 32
Any	1 / 3	29 / 39	44 / 48	40 / 51	78 / 89

Table 4: Safe/Unsafe flow summary of 78 flow scripts

# Safe	**# Unsafe** (54)		**# Both**	**# Mix**
	# Unvetted	**#Tampered**		
45	40	47	21	0

analysis detects that 11 scripts have no information flows. Thus, using potential flow (presence of both source and sink), reduces prompting by 48.26% (from 172 to 89) over the traditional capability approach (presence of sources). Using actual information flows, as computed by our analysis, further reduces prompting by 12.36% (from 89 to 78).

Table 3 shows the information flow summary of the published scripts based on source-sink pairs. Each column represents a kind of sink and each row represents a kind of source. The first number in each table cell is the number of scripts for which our analysis determines information flow from the given source to the given sink, whereas the second number in each cell is simply the number of scripts that use the corresponding source and sink. The two numbers presented in each table cell compare our approach of computing actual information flow, to a naïve capability analysis that simply presumes an information flow for each used source-sink pair.

For example, the cell for Camera and Web shows that a naïve capability approach would classify 30 scripts as having information flow from the camera to the web, whereas our information flow analysis proves that none of these scripts actually leak camera information to the web, completely removing the concerns of leaking pictures taken from the camera through the web.

Similarly, the naïve capability approach would consider 19 scripts to leak contact information through the web, while our analysis shows that none of these scripts would do that.

These results show that information flow analysis effectively computes a much finer granularity of the potential flows between sources and sinks used in a script.

7.2.2 RQ2: Safe Scripts

To address RQ2, we apply our static analysis on the 78 subject scripts that have information flows, referred to as *flow scripts*, and measure the number of flow scripts that have safe flows. We assume only sink *Web* is an unvetted sink, while all others are vetted sinks. Table 4 shows the safe/unsafe flow summary of the 78 scripts that have information flows. Column "# Safe" shows the number of scripts that have safe flows. Column "# Unvetted" shows the number of scripts that have information flows from sources into unvetted sinks. Column "# Tampered" shows the number of scripts that have tampered information flows. Column "# Both" shows the number of scripts that have both safe and unsafe flows. Column "# Mix" shows the number of scripts that have both safe and unsafe flows from a common source (*mix scripts*).

The results show that 45 (57.69%) flow scripts have safe flows and 54 (69.23%) flow scripts have unsafe flows. Among these 54 unsafe flow scripts, 40 flow scripts have flows from sources into unvetted sinks and 47 have tampered information flows. Based on this safe/unsafe flow summary, we know that 24 (#Safe−#Both), or 30.77% of flow scripts have only safe flows. For these 24 scripts, users are perfectly safe to use the scripts granting full access to private information without prompting or reduced functionality.

Table 5: Categorization of sources

	Naïve	**Flow**	**Safe**	Unsafe due to		
				Unvet.	**Tamp.**	**Both**
Camera	36	33	24	0	9	0
Contacts	41	30	25	0	5	0
Location	34	30	0	27	26	23
Microph.	6	1	1	0	0	0
Music	3	1	0	0	1	0
Picture	32	24	6	14	15	11
Total	152	119	56	41	56	34

Among the 21 flow scripts that have both safe and unsafe flows, none are mix scripts. In all the TouchDevelop scripts, only 2 flow scripts published by ourselves have both safe and unsafe flows from a common source to sinks. Our current access granting allows users to grant access based on sources only, instead of flows. Users cannot choose real information for one flow and anonymized information for another flow from the same source. As we only found 2 scripts where this limitation matters, it seems to be a good trade-off that avoids giving users too much choice.

7.2.3 RQ3: Safe Sources

To address RQ3, we look at how many times a user would have to change the default setting for a source if she were to give full access to all scripts. Table 5 shows the total number of times a source appears in a given context. Column "Naïve" shows the number of scripts that use this source and any sink. Column "Flow" shows the number of scripts that have information flows from this source to any sinks. Column "Safe" shows the number of scripts for which this source is safe. The last three columns explain why some flows are unsafe. Column "Unvetted" shows the number of scripts where information flows from this source to unvetted sinks. Column "Tamper" shows the number of scripts where information from this source is tampered before it reaches a sink. Column "Both" shows the number of scripts that have common sources in Columns "Unvetted" and "Tamper".

Among 33 scripts that have source Camera appearing in flows, 24 scripts (72.73%) have source Camera as a safe source and 9 scripts (27.27%) have source Camera in tampered flows. Similarly, 25 scripts (83.33%) have safe sources of Contacts, leaving only 5 scripts having source Contacts appearing in tampered flows.

In summary, our analysis detects that 47.06% (56) of 119 sources are safe sources. These safe sources are allowed to use real information directly based on our default settings, eliminating the need for access granting. Among the remaining 63 unsafe sources (# Unvetted + # Tamper - # Both), 7 (#Unvetted - #Both) are solely due to flow to unvetted sinks, and the remaining 56 sources appear in tampered information flows. These results show that using the naïve classification, a user would have to make 152 changes to settings to use real data in all scripts. Using information flow alone, this number is reduced to 119 changes. Using tamper analysis and vetted sinks in addition to information flow, our approach reduces the burden to 63 changes to settings, an overall reduction of 58.6%.

8. DISCUSSION AND FUTURE WORK

In this section, we discuss generalizations and limitations of our approach.

Generalization to Other Mobile-Device Platforms. To generalize our approach to other mobile-device platforms, such as

Windows Phone, Android, and iOS, several points need to be addressed: 1) these platforms provide a much larger API surface than TouchDevelop and annotating these APIs with source, sink, and flow information is a major effort, 2) the languages used (Java, C#, or assembly code) provide more ways to obscure flow than in our scripting language, in particular through indirect calls, or via reflection. The static analysis would have to be extended to account for these [13,14]. 3) Indirect flow through mutable storage will require a finer grained heap model than we currently employ (one abstract location per data kind). The static analysis might need to be complemented with dynamic analysis [1,15] to address this issue.

Limitations of Static Information Flow Analysis. Due to the way our approach handles implicit flows, our approach may produce false positives as described by Kang et al.'s work [16]. However, our evaluation results show that even with these potential false positives, our approach still achieves a significant reduction in access granting for users. To improve our approach when migrating to other mobile-device platforms, our approach can be combined with DTA++ techniques [16].

Another type of implicit flow, covert channels [17], may cause false negatives of our approach. For example, a script can store a classified picture into the media library, and then later share it through facebook via a different application. Our flow analysis would indicate that a picture is stored into the media library (and the user has to agree with that flow), but our approach does not contemplate what could happen to the picture in the library after that. To address such issues, the operating system would have to provide dynamic taint tracking [1], since such flows involve more than one application or even OS built-in functionality.

9. RELATED WORK

In this section, we compare our work with other related approaches.

User-Aware Application Capabilities. Mobile-device platforms like Android and social-network platforms like Facebook use manifests to show application capabilities and request permissions at install time. Other mobile-device platforms like iOS and research approaches like TaintDroid [1] report application capabilities the first time applications try to access a resource. The capabilities shown in the manifests are either claimed by developers [18] or only present part of the requested application capabilities. Felt et al. [14] proposes an approach that uses static analysis to map API calls used by applications to permissions, which is similar to our approach. However, they adapt automated testing methodology to test the applications and identify APIs that require permissions, while our approach annotates the APIs with permissions and uses static checking.

Information Flow Analysis. Xie and Aiken [19] present an approach that statically computes summaries of blocks and procedures of PHP and detects security vulnerabilities at the block level, intraprocedural level, and interprocedural level. 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 $\rightarrow$ take camera picture) that cannot be analyzed by their approach. To address these problems, our approach uses mutable locations to simplify analysis of reference-type flows and tracks untampered- and tampered-classified information for classifying safe and unsafe flows.

The closest work related to ours is PiOS [20], which studies private information leakage in actual iOS binaries. The PiOS approach statically computes data flow along control flow paths from sources to sinks to determine if there exists a user prompt along that path. PiOS emits warnings if such a flow is found without a user prompt. For the purposes of safe-guarding users of the TouchDevelop bazaar, the PiOS approach is insufficient because: 1) the PiOS analysis is not conservative, it misses flows that are too long or use indirect flow, 2) the prompts PiOS identifies may be unrelated, show nothing of the leaked information, or show tampered information. PiOS also does not use the static information to control user prompting and privacy settings as our approach does.

Language-based information flow [21] allows developers to annotate variables with security attributes. These attributes can be used by compilers to enforce information flow controls. For example, Slam [22] shows that information flow labels can be applied to a simple language with reference types and Jif [23, 24] extends Java language with statically-checked information flow annotation. Laminar [25] 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, especially for beginners.

Dynamic taint analysis [1, 15] 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, providing accurate runtime information about leaks. However, to reduce runtime overhead, these approaches usually ignore implicit flows raised by control structures. Moreover, dynamically executing all execution paths of these applications to detect potential information leaks is impractical. These limitations make these approaches inappropriate for computing information flows for all submitted applications.

Access Granting. Mobile-device platforms like Android and social-network platforms like Facebook use manifests to request permissions at install time. Once permissions are given by users, the permissions cannot be changed. iOS and Windows User Account Control [26] prompts a dialog to request permissions from users when applications try to access a resource or make security or privacy-related system-level changes. Instead of only showing information about the access to resources, our approach presents information flows to describe what applications may do with private user information. Our access granting also provides a way for users to try out applications before using private information, and these settings can be changed at will.

Zhu et al. propose an approach that uses dynamic taint analysis to track user data as it flows through applications [15]. Their approach allows users to choose among logging the action, blocking the system call, or randomize the tainted data. Chen ea al. also propose an approach that shadows data that the user wants to keep private and blocks network transmissions that contain data the user made available to the application for on-device use only [27]. Our anonymized/real/abort setting is inspired by their approach, but we use static information flow analysis extended with tampered information to classify flows as safe/unsafe flows and provide default access settings, rather than runtime information.

Automated Security Validation of Mobile Apps. Gilbert et al. present a vision of making mobile apps more secure via automated validation [2]. 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 graphs [28] or natural language processing. Such an approach is akin to automated testing and suffers from the same problems, namely coverage. It is difficult to drive applications automatically into exercising all data and control paths. Thus, in the end, such an approach only gives a partial view of the behavior and does not safe-guard users.

10. CONCLUSION

We presented a user-aware privacy control approach based on static information flow analysis extended with tamper analysis. We compute information flows from private sources to sinks and classify them as safe/unsafe flows. We conducted evaluations on 546 scripts published in TouchDevelop to study the effectiveness of our static information flow analysis. The results show that our approach computes useful information flows and can be used to automatically provide default privacy settings for each script that keeps users safe without any user intervention, thereby obviating the need for manual script validation.

Our approach is the first step towards employing a better privacy control mechanism in mobile-device platforms based on automatic validation of applications in the marketplace and user-driven access control.

11. REFERENCES

[1] 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.

[2] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps At App Markets. In *Proc. MCS*, pages 21–26, 2011.

[3] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The Effectiveness of Application Permissions. In *USENIX Conference on Web Application Development (WebApps)*, 2011.

[4] T. Vidas, N. Christin, and L. Cranor. Curbing Android Permission Creep. In *Proc. W2SP*, Oakland, CA, May 2011.

[5] Franziska Roesner. User-Driven Access Control: A New Model for Granting Permissions in Modern Operating Systems. *Qualifying Examination Project, University of Washington*, June 2011.

[6] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Programming Languages and Systems*, volume 6012 of *LNCS*, pages 64–84. Springer, 2010.

[7] TouchDevelop. http://research.microsoft.com/TouchDevelop.

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

[9] Fraser Howard. Malware with your mocha: Obfuscation and anti-emulation tricks inmalicious JavaScript, September 2011. http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf.

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

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

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[13] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proc. of USENIX Security Symposium*, 2011.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proc. CCS*, 2011.

[15] 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.

[16] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. of NDSS*, San Diego, CA, February 2011.

[17] Shiuh-Pyng Shieh and Virgil D. Gligor. Auditing the use of covert storage channels in secure systems. In *IEEE Symposium on Security and Privacy*, pages 285–295, 1990.

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

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

[20] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS : Detecting privacy leaks in iOS applications. In *Proc. NDSS'11*, 2011.

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

[22] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy And Integrity. In *Proc. POPL*, pages 365–377, 1998.

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

[24] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. POPL*, pages 228–241, 1999.

[25] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. Mckinley, and Emmett Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proc. PLDI*, pages 63–74, 2009.

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

[27] Yan Chen, George Danezis, and Vitaly Shmatikov, editors. *Proc. CCS*. ACM, 2011.

[28] 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.