Poster: Maxwell: User-Driven Information Flow Control for Android

Jun Yuan (Student)
Computer Science Department
Stony Brook University
Stony Brook, New York 11792
Email: junyuan@cs.stonybrook.edu

Rob Johnson (Faculty)
Computer Science Department
Stony Brook University
Stony Brook, New York 11792
Email: rob@cs.stonybrook.edu

***Abstract*—Current mobile operating systems protect users' privacy by enforcing access control policies on sensitive data sources and sinks. However, many privacy goals are really information flow problems. We present *Maxwell*, a system that enables users to explicitly authorize information flows as part of their normal interactions with applications. Our work extends the ideas of user-driven access control and applies them to information flow control. Our implementation uses TaintDroid to enforce the user's information flow control choices. Our implementation represents information flow privileges using an object-capability model which may be of independent interest.**

I. Introduction

Modern operating systems use several different methods to decide whether to give an application access to a privileged resource. For example, systems may silently approve some accesses, ask users to approve application permissions at install time, or ask users to approve accesses at run-time[2]. Researchers have also proposed trusted user interfaces for integrating permission granting actions into the user's normal workflow[4]. However, trusted user interfaces are not always applicable and the other methods impose a trade-off between usability and the precision of the policy that is enforced.

This poster proposes a new point in the design space. We present *Maxwell*, a system that enables users to explicitly authorize information flows as part of their normal interactions with applications.

Maxwell advances the state of the art in three ways.

First, by enforcing information flow controls, *Maxwell* reduces the impact of granting access to privileged data sources, such as a GPS receiver, or sinks, such as the network. For example, in current systems users cannot give an app access to the GPS and network without also giving it permission to transmit their GPS location over the network. Consequently, the operating system cannot silently grant an application permission to the GPS and network – the user must approve these accesses. If, on the other hand, the system enforces an information flow policy that forbids GPS data from being transmitted over the network, then the system can silently grant applications access to the GPS without harming user privacy. This improves usability with no sacrifice in privacy.

Second, *Maxwell* supports user-driven information flow control, giving users fine-grained control over how their information is used without imposing an undue usability burden. Like user-driven access control, user-driven information flow control merges permission-granting into the user's normal workflow. For example, when a user clicks a button to share a photo over the internet, the application simultaneously receives the permissions needed to transmit camera data over the network. This enables users to manage fine-grained permissions with little cognitive load or distractions and without succumbing to dialog fatigue.

User-driven information flow control can enable systems to side-step many of the limitations of trusted user-interface widgets. For example, a trusted user-interface widget for taking photos should include a secure preview window. However, moving the preview window into a separate, trusted process is incompatible with photography applications that want to apply custom filters to the camera preview image. With user-driven information flow control, the OS can allow the application to access the camera directly. The application can't release the photo over the internet, though, until the user invokes the trusted user interface component for sharing photos.

Finally, *Maxwell* represents applications' information flow privileges using an object-capability design. Our system uses a modified version of TaintDroid to enforce a default-deny information flow policy: any data item that is tainted as coming from any sensitive source is not allowed to be written to any sensitive data sink. When a user activates a trusted user interface component, the application is passed a *Declassifier* object with a *declassify* method. The application can use the Declassifier to obtain "declassified" references to objects, which can then be written to sensitive sinks.

The object-capability design offers several advantages. First, it separates the act of granting information-flow privilege from the act of using it. This flexible approach supports a wide variety of application designs – applications can obtain additional user input, perform computation, data marshalling, etc., before actually using the declassification capability.

This design also supports flexible information flow policies. For example, a trusted user interface button can grant the application a one-time-use declassifier object, whereas a trusted toggle switch can give the application a declassifier that can be called repeatedly until the user disables the privilege, at which time the declassifier will be deactivated. Declassifiers can also be tailored to specific sources and sinks, e.g. a photo sharing button would pass the application a one-time declassifier that will only declassify camera data so that it can be transmitted over the network. The declassifier would reject an attempt to declassify other types of data or to pass camera data to a different type of sink.

Finally, declassifiers bring all the standard benefits of the object-capability model. The application can control which of its components can use declassifier capabilities, unlike with an ambient-authority-based implementation. For example, a benign application that embeds an aggressive advertising library can choose whether to give the library code access to declassifiers. It can also delegate restricted declassification capabilities to its sub-components by implementing wrapper declassifiers.

II. Prototype Implementation

We developed a *Maxwell* prototype for Android. Our implementation uses TaintDroid to enforce information flow policies. We have implemented a declassifier API and several trusted widgets. This section describes the key design decisions of our implementation.

First of all we extend TaintDroid to support fine-grained source to sink flow control. TaintDroid implements an efficient system-wide information flow tracking that tracks the source of each byte of the data as it flows through the execution stacks [1]. However, TaintDroid does not block private data at sinks, rather it just identifies the data and logs a warning message.

- **Taint tag extension**. A taint tag is a 32 bit vector attached to each variable. By default, only lower 16 bits have been used for tracking different taint sources such as GPS and contacts. We assign the upper 16 bits of taint tag to represent different sinks(e.g., network).
- **Sink checking**. We modify the TaintDroid code at all the sink points to check if the corresponding sink bit of taint tags is cleared, otherwise an exception would be thrown to block the data path.
- **Declassifier API**. We implement *Declassifier* interface in the *libcore.dalvik.system* package for applications to create a declassified copy of tainted data from the given policy. Internally, *Declassifier* implements private helper routines as Dalvik native code to clear taint bits. *Declassifier* is the only class that can clear the taint bits so it is made only package accessible. Only trusted classes in the same package are allowed to use it. *Declassifier* API supports a variety of policies which can be represent in a triple $(source, sink, mode)$ in which $mode$ could be *single-use*, *unlimited* or *timed* for different access semantics. *Declassifier* object can keep the access control state(e.g., elapsed time) to check against its policy. If its access control state fails the policy checking(e.g., timer is up), the *Declassifier* returns null to all untaint calls to assure the privilege is revoked.

Declassifier is designed and implemented to represent the capability of clearing taint bits. Applications can clear taint bits of their data only when getting a hold of a *Declassifier* instance. The only way for an application to access a *Declassifier* object is to request a *TrustedWidget* object through *ResourceMonitor*.

ResourceMonitor in *Maxwell* is a privileged final static class that lies in the same package of *Declassifier* and exclusively manages access to the capability (*Declassifier*). Unlike RM in User-driven access control, our *ResourceMonitor* is relieved from managing access control state which is shifted to *Declassifier*. The *ResourceMonitor* exposes to applications *TrustedWidget* instances tied with capabilities.

TrustedWidget is the implementation of trusted user interface by extending android widget class and restricting the access to the constructor and other unsafe methods of the widget. A *TrustedWidget* is coupled with its *policy* and capability granted by *ResourceMonitor*. We will walk through how *ResourceMonitor* bridges between *TrustedWidget* and *Declassifier* API.

- An application requests *ResourceMonitor* for a *TrustedWidget* object by providing a *policy* object in the form of $(source, sink, mode)$ and a *callback* object that defines *OnClickWithCap* method.
- *ResourceMonitor* constructs a *TrustedWidget* instance based on the parameters. The drawable of the *TrustedWidget* object is rendered without ambiguity based on the *policy* so that users understand what the information flow choice the widget represents.
- *ResourceMonitor* constructs a *Declassifier* instance based on the *policy* and registers the OnClickListener of the *TrustedWidget* object to be using *OnClickWithCap* called with the *Declassifier* instance.
- The *TrustedWidget* instance is returned to the application. The application can now use the *Declassifier* to untaint the data. If the access control state of the *Declassifier* is valid to pass the policy check, *Declassifier* creates an declassified copy of the user data from the given policy.
- The declassified copy of the data flows to the sink points and only if the taint tag of the data passes the sink checking, the data can leave the device.

III. Preliminary Result and Future Work

We have developed applications using trusted widgets of *Maxwell* to test the policies with *single use*, *unlimited* and *timed* modes respectively. The contrast test of regular widgets versus trusted widgets indicates that the user data can be sent over to the designated sink only with the user triggering the trusted widgets and the given policies are loyally enforced.

Maxwell inherits two issues from TaintDroid tracking: indirect control flow and native code of applications. Besides, the object-capability model of *Maxwell* works based on the assumption that the application code does not violate Joe-E capability model [3] (e.g., no reflections). The improvements on these limitations are in progress.

References

[1] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[2] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *7th USENIX Conference on Hot Topics in Security*, 2012.

[3] A. Mettler, D. Wagner, and T. Close. Joe-e: A security-oriented subset of java. In *Network and Distributed System Security Symposium*, 2010.

[4] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.