PhoneWrap - Injecting the "How Often" into Mobile Apps

PhoneWrap - Injecting the "How Often" into Mobile Apps

Daniel Franzen
LFCS, University of Edinburgh
D.Franzen@ed.ac.uk

David Aspinall
LFCS, University of Edinburgh
David.Aspinall@ed.ac.uk

Abstract

Mobile apps have access to a variety of sensitive resources and data. Current permission-based policies guarding these resources are not expressive enough to distinguish the wanted functionality from malicious attacks. We present the tool *PhoneWrap* which inserts fine-grained ticket-based policies into mobile JavaScript apps written with the *PhoneGap* framework. Our policies grant a bounded number of accesses for each functionality based on the user's interaction with the app. The policies are enforced without modification of the execution environment. We have applied PhoneWrap successfully to hand-crafted examples and real-world Android apps to show that accurate policies can be retrofitted.

1 Introduction

Modern mobile devices have access to private data, sensors and services. Mobile operating systems like Android and iOS govern access with permissions which that are either fully granted ahead of time, or can be switched on or off during use in limited ways. Neither mechanism allows precise fine-grained control on how often or in which context a granted resource may be used. But when users notice resources being overused in the wrong context, they react. A user of a permission usage monitoring app complained: "Why would WhatsApp access my contacts over 7000 times [...]. It should only access my contacts when I open the app. Deleted that app is what I did."[9] A policy based on the least-privilege principle should specify how often and in which context a resource can be accessed.

At the same time, we see a rise of JavaScript as a programming language for mobile devices. Mobile operating systems like Tizen and ChromeOS building directly on JavaScript are emerging, but frameworks, such as PhoneGap [1], utilising a built-in browser to execute JavaScript and HTML apps on established operating systems are widely adopted by developers already.

In this paper, we introduce *PhoneWrap*, a tool to provide more expressive policies for JavaScript apps. PhoneWrap manages a set of *tickets* via an inline reference monitor; it requires one ticket for each resource access. Tickets are either granted at launch or generated according to the user's interaction with the app to allow user-requested functionality. Special *local* tickets are only valid during the processing of a specific user event. If access is requested without tickets, PhoneWrap executes a denial behaviour.

1.1 Wrapping

PhoneWrap enforces its policies by "wrapping" the resource-consuming APIs, inspired by Phung et al's Self-Protecting JavaScript [15]. Each resource-consuming function is provided with a a resource-counting instrumented version, and references to the original are replaced. Subsequently, application code can only access the instrumented API which, depending on the policy state, either calls the original API or a deny behaviour. To isolate the original function and the policy state from the application code, the policy is implemented inside a function scope in which PhoneWrap stores the policy state and the original resource consuming APIs as local variable. The JavaScript scope mechanism ensures that local variables cannot be accessed from outside the function body; this ensures that the policy is enforced.

In: D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer (eds.): Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS'16, London, UK, 06-April-2016, published at http://ceur-ws.org

1.2 Motivating example

The app "TrackMyVisit" (Figure 1) manages a list of journeys. While a journey is active, the app logs the GPS position and sends a message to up to 3 specified contacts in case of an emergency. Furthermore the app can store pictures for each journey. This functionality requires Android permissions to take pictures, to access the GPS position and to send messages. However, these permissions could be abused to invade the user's privacy, to send personal data, to impersonate the user or to charge for premium messages. The permissions model is not fine-grained enough to deny this potential malicious behaviour while preserving the expected functionality. PhoneWrap can enforce that the GPS sensor is only accessed while a journey is active. It grants exactly one camera ticket for each time the "Add Image" button is pressed and 3 message tickets for the "Emergency" button. The latter tickets are marked as local and are cancelled once this emergency routine has been performed to ensure that unused tickets cannot be abused later.

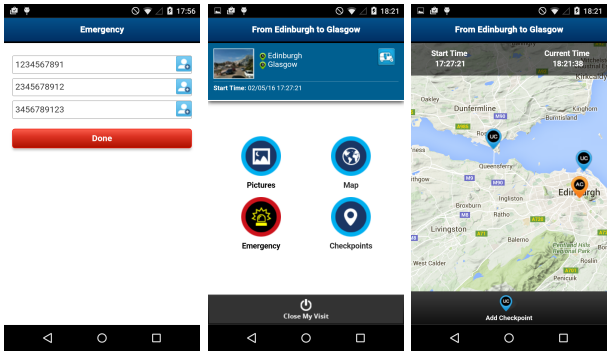

Figure 1: TrackMyVisit (myzealit.TMV.apk)

1.3 Contribution

This paper presents the following:

- a formalisation of interaction-dependent ticket-based policies;
- the PhoneWrap system, which semi-automatically injects a policy into PhoneGap apps;
- a small-scale evaluation of PhoneWrap.

Previous research has established that apps are often over-privileged [3] and use resources differently than the user expects [11]. (For more related work, see Section 5.) PhoneWrap fixes both issues by enforcing explicit resource bounds. The enforced policies have the following features:

- Access can be granted or revoked according to user interaction.
- Each resource can be restricted to a finite number of accesses.
- A violating access can be replaced by any JavaScript definable function.

Previously, inlined reference monitors were used to implement different policies for JavaScript [15] and Android apps [2]. Quantitative policies [6] or interaction-dependent policies [18] have been studied separately. PhoneWrap is, to the best of our knowledge, the first system which enforces quantitative interaction-dependent policies for JavaScript apps in an unmodified execution environment.

The remainder of this paper is structured in the following way. Section 2 will discuss the interaction-dependent ticket-based policies. Section 3 presents an overview of the PhoneWrap system. Section 4 presents the results of the evaluation on real-world apps and Section 5 discusses related work. Finally, Section 6 concludes by discussing the results and limitations of the current system.

2 Interaction-dependent Ticket-based Policies

PhoneWrap enforces bounds on the resource consumption based on tickets, each "paying" for one-time access to the guarded resource. Tickets are granted at launch and for specific UI events to allow the functionality activated by the interaction. Tickets can be specified as local to a UI event to limit their scope this event. Unused local tickets are cancelled after all event handlers for that event have been executed.

2.1 Policy Model

The policies considered here react to API calls and user events $e \in \mathit{UIEvents} = \{\texttt{click}, \texttt{mousedown}, ...\}$ including the special event $\texttt{start} \in \mathit{UIEvents}$. JavaScript's "Run-to-completion" model guarantees that each user event e is handled before the next user event is executed. The properties of the policies consider the resource consumption of *traces*.

Definition 2.1. 1. Let a R be the set of resource accessing APIs and

$$RM : API \to \{0,1\} : RM(f) = 1 \Leftrightarrow f \in R$$

be called the *resource model*.

2. Let BP be a set of button policies. Each $bp \in BP$ consists of a triple $(cond, mperms, local)$, indicating $mperms$ tickets will be generated for each event matching the condition $cond$. The flag $local$ of bp indicates whether the generated tickets is local to the event.

3. From a set of button policies BP, the policy $pol = (pol_l, pol_g)$ is a pair of functions that describes the

local and global tickets generated by each event, defined as:

$$\begin{aligned} pol_l(e) &= \sum_{\substack{bp \in BP \\ bp.local \\ bp.cond(e)}} bp.mperms \\ pol_l(start) &= 0 \\ pol_g(e) &= \sum_{\substack{bp \in BP \\ \neg bp.local \\ bp.cond(e)}} bp.mperms \\ pol_g(start) &= m_0 \end{aligned}$$

where m_0 is the number of tickets granted by PhoneWrap at launch.

Definition 2.2. 1. We distinguish the following *policy events*:

(a) $API(f)$: the API f is called

(b) $E(e)$: the event $e \in UIEvents$ occurred

(c) $Done(e)$ the event e has been processed

2. We define a *trace* as the possibly infinite sequence $w_1, w_2, ...$ of policy events occurring during an execution of an app P.

3. For each finite prefix $w_1, ..., w_k$ of a trace w define the *resource count* c_{res} by:

$$c_{res}(w_1, ..., w_k) = \sum_{w_i = API(f)} RM(f)$$

4. Let $w|_e$ be the sub-trace $E(e), ..., Done(e)$ of w.

5. Define the *ticket count* c_{tic} of a trace $w_1, ..., w_k$ recursively as

$$\begin{aligned} c_{tic}(\epsilon) &= 0 \\ c_{tic}(w', API(f)) &= c_{tic}(w') - RM(f) \\ c_{tic}(w', E(e)) &= pol_l(e) + pol_g(e) + c_{tic}(w') \\ c_{tic}(w', Done(e)) &= \min(0, c_{res}(w|_e) - pol_l(e)) \\ &\quad + c_{tic}(w') \end{aligned}$$

with $w = (w', Done(e))$ in the last case

Intuitively, c_{tic} tracks the number of available tickets. It subtracts tickets for each API call and adds them if an event occurs. After an event is handled, it subtracts local tickets if they have not been used within the event scope. Not that due to JavaScript's "Run-to-completion" model event traces cannot be nested.

We say a trace $w = w_1, w_2, ...$ *conforms* with the policy *pol* if there is no i such that $c_{tic}(w_1, ..., w_i) < 0$.

Definition 2.3. Given a trace w and a policy *pol*, let the *enforced trace* $w|_{pol}$ be the longest conforming prefix of w:

$$(w_1, w_2, ...)|_{pol} = w_1, ..., w_i$$

where i is the smallest index with $c_{tic}(w_1, ..., w_{i+1}) < 0$ and

$$(w_1, w_2, ...)|_{pol} = w_1, w_2, ...$$

if no such i exists.

From this definition some desirable properties for a policy follow:

1. The resource consumption of a conforming trace $w = w_1, w_2, ...$ is bounded by the policy: $c_{res}(w) < \sum_{w_i = E(e)} pol_l(e) + pol_g(e)$.

2. Since $w|_{pol}$ conforms with *pol*, the enforced trace is bounded by this bound.

3. Functionality conforming with the policy is preserved: $w|_{pol} = w$ if w conforms with *pol*.

2.2 Policy Specification

A PhoneWrap policy consist of 3 parts: (1) the guarded resource (2) the button policies (3) the deny behaviour.

PhoneWrap guards services (phone calls, messages or social network interaction), sensors (microphone, camera, GPS location) or content (contact and calender data, conversations, documents, pictures). A resource is specified by its APIs, for example, the API `smsplugin.send` which sends SMS messages. This determines R and therefore RM.

The button policies in PhoneWrap are defined as the triples $(cond, mperms, local)$. The condition *cond* is defined by the HTML properties of the target element of the event. In the "TrackMyVisit" example, the button with the icon path `src` ending in `images/emergency_icon2.png` generates 3 tickets for each click.

Finally, the policy defines the deny behaviour which is executed in case of an attempted policy violation. It is defined as a JavaScript function with access to the policy state. This enables many possible reactions adjusted for the guarded resource, e.g., terminating the app, ignoring the resource request, returning dummy values or inquiring with the user. To match the formal definition of enforced traces, the deny behaviour is set to `exit(0)` to terminate the execution on policy violation.

PhoneWrap policies include many other classes of policies. By granting infinitely-many local/global tickets for each user interaction, PhoneWrap can enforce "no resources in the background" and "only after first interaction" policies. With the deny behaviour set to display a confirmation dialog and to grant 1 or ∞ many tickets when confirmed the policy is equivalent to OneShot or Session permissions in J2ME or the iOS operating system. PhoneWrap can also deny access completely like the Android system when the corresponding permission is missing.

3 The PhoneWrap system

The PhoneWrap system has been developed with the following core aims:

Usability PhoneWrap is aimed at users with a basic understanding of the user interface and resource behaviour of the guarded app and the ticket-based policies. We do not assume knowledge of the app's source code or the PhoneWrap's enforcement method.

Stand-alone PhoneWrap is contained within the guarded app. Modifications of the execution environment or the Android system require root access to the phone which undermines important security principles and many users are not able or willing to make this modification.

Real-world The enforcement of PhoneWrap has to work on apps executable on real mobile phones.

An app is fitted with a policy in 5 steps (see Figure 2): (1) *unpacking*, (2) *information extraction*, (3) *policy creation*, (4) *injection* and (5) *repackaging*. For steps (1) and (5), PhoneWrap uses the available tools `adb`, `zip` and `apktool`. Unfortunately, the policy injection invalidates the developer signature of the original package. PhoneWrap resigns the modified package with a new key, the *policy key*. Assuming the policy creator verifies the original signature, this new policy signature certifies the integrity of the original package and the injected policy. In isolated cases the signature can be vital to the functionality of the app. Apps that were signed with the same key originally should be signed with the same key after policy injection such that inter app communication is preserved. In isolated cases, for example for the Google Maps web API, a license can be linked to the signature. In this case, the policy creator would need to obtain a new license for the policy key.

For the *information extraction*, the PhoneWrap script `m20_analyse` finds the Android manifest and the PhoneGap configuration file according to the package layout. From these files PhoneWrap extracts the requested permissions and PhoneGap plugins used to access the resources as well as the main source file and source folder. The *injection* step inserts the PhoneWrap enforcement script into this main HTML folder and links it into the main HTML file using the HTML `src` tag.

The *policy creation* requires human action. The policy creator has to identify the guarded APIs from the extracted permissions and plugins and specify the UI elements for the button policies. PhoneWrap assist the latter by instrumenting all buttons in such a way that they show their properties during the normal interaction with the app. To check a given policy PhoneWrap can highlight all effected UI elements. The deny behaviour is specified as a JavaScript function. The whole policy can be written in JavaScript or using PhoneWrap's HTML form shown in Figure 3 which embeds the policy parameters into full English sentences.

3.1 Policy enforcement

PhoneWrap provides the wrapper script which contains the policy and the code to wrap the necessary functions according to [15]. For each call to the critical APIs, PhoneWrap decreases one of the two counters `mperms_local` and `mperms_global` which indicate the number of available local / global tickets. Local tickets are used with preference. If no ticket is available, PhoneWrap calls the specified deny behaviour instead. Additionally, PhoneWrap listens for UI events at the root of the DOM-tree. Since every event is first evaluated at the root node, PhoneWrap receives all UI events this way. When PhoneWrap receives a matching event, it uses `setTimeout` with 0 seconds to insert a callback into the HTML event handler queue behind all handlers for this event. As a consequence, this callback is executed after all handlers have finished and cancels all remaining local tickets.

All code necessary for the enforcement is contained in one JavaScript file which is inserted into the header of the main HTML file. When executed, the file immediately wraps all available critical APIs.

Most PhoneGap APIs to access the resources are provided as plugin and implemented as Java class. Each plugin provides a JavaScript API which internally calls its Java API using the PhoneGap bridge `exec`. Additionally, PhoneGap uses the `require` function to obtain the JavaScript API. PhoneWrap wraps the JavaScript API but also wraps the functions `exec` and `require` in case the application code calls them directly. The wrapped versions of these functions check whether the requested API accesses the guarded resource and enforces the policy.

Different PhoneGap APIs become available at different times in the app's life cycle. Some are available from the start, others after the PhoneGap library has been loaded or initialised and some are defined in a separate source file. PhoneWrap uses JavaScript methods to recognise these situations and immediately wraps all available APIs. Since it is not trivial to determine whether an API has been wrapped already, PhoneWrap re-wraps all APIs multiple times. For example, if the function `smsplugin.send` has already been wrapped and a new source file is loaded into the app, PhoneWrap re-wraps `smsplugin.send` since the

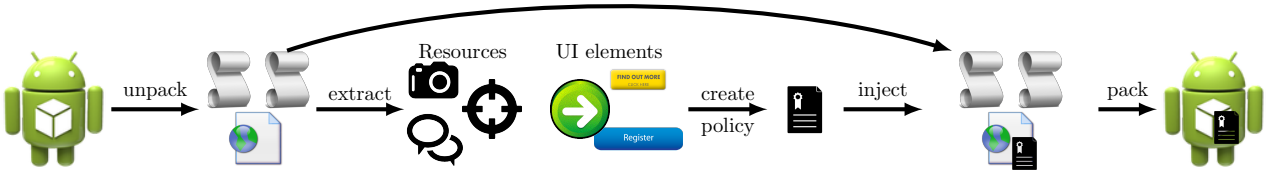

Figure 2: Policy injection

Figure 3: Policy Creation Form

new source file could have redefined this API. This behaviour might result in a second wrapping layer around `smsplugin.send`. However, in this case the outer layers do not subtract additional tickets, which ensures that each API call is only paid by 1 ticket. As final fortification, PhoneWrap prevents the app from generating additional tickets by simulating user events.

3.1.1 Policy security properties

Claim 3.1. *The application code cannot change the state of the policy.*

Justification. JavaScript protects the local variables of the policy function, including the policy state, from access outside the policy function.

Claim 3.2. *The application code cannot access the original API directly.*

Justification. The PhoneWrap script is inserted at the top of the header of the main HTML file. Therefore, it is executed before any other scripts and overwrites all API references specified in the policy. The original references are protected like the policy state. In particular, JavaScript's dynamic scoping means that also all libraries internally calling the resource API only have access to the wrapped API.

Claim 3.3. *The tickets for a UI element are produced before the resource consuming action occurs.*

Justification. The PhoneWrap script is executed first and registers the first listener at the root node. Each event is first passed to the root node and the JavaScript standard executes event handlers in the order of registration.

Finally, the policies enforced by PhoneWrap implement the formal model of interaction-dependent ticket-based policies described earlier:

Lemma 3.4. *Given the definition of RM and pol_g, pol_l in Section 3, the PhoneWrap counters `mperms_global`+`mperms_local` after execution of the trace w are equal to $c_{tic}(w)$.*

Proof. This can be proven by induction on the length of the trace w. The only interesting case is $w = (w', Done(e))$:

- Assume after w' the counters are equal to $c_{tic}(w')$.
- $Done(e)$ means PhoneWrap sets the counter `mperms_local` to 0.
- $c_{res}(w|_e)$ is the resource consumption of the the scope of the event e. $\Rightarrow$ During the execution of $w|_e$ PhoneWrap has decreased the counters (with preference on `mperms_local`) $c_{res}(w|_e)$ times.

If $c_{res}(w|_e) \geq pol_l(e)$, PhoneWrap has already reduced `mperms_local` to 0, so the event $Done(e)$ does not change the counters. By the definition of c_{tic} it follows $\min(0, c_{res}(w|_e) - pol_l(e)) = 0 \Rightarrow c_{tic}(w) = c_{tic}(w')$. Otherwise, if $c_{res}(w|_e) < pol_l(e)$, `mperms_local` has $pol_l(e) - c_{res}(w|_e)$ tickets left to cancel. Similarly $\min(0, c_{res}(w|_e) - pol_l(e)) = -(pol_l(e) - c_{res}(w|_e))$ and therefore $c_{tic}(w) = c_{tic}(w') - (pol_l(e) - c_{res}(w|_e))$. □

Claim 3.5. *PhoneWrap implements a ticket-based policy for the default-deny behaviour `exit(0)` (terminate execution).*

Justification. This claim follows from the preceding Lemma and the fact that the deny behaviour `exit(0)` replaces the first API call resulting in a non-positive ticket count with the terminating statement equivalent to $w|_{pol}$.

PhoneWrap can also define more complex deny behaviours. This makes it possible to execute parts of policy-violating apps. With different deny behaviours the claimed properties still hold. However, the behaviour of violating traces might be changed arbitrarily if the continuation after a policy violation depends on the result of the violating action.

PhoneWrap can be used to inject *multiple policies* into the same app to restrict the use of multiple resources or policies of different parties (developer, distributor, user). PhoneWrap inserts multiple wrapper scripts into the DOM tree, each executing the previously injected wrapper if its own policy is fulfilled. The application code only has access to the outermost wrapper and the original API is only called if all injected policies accept the behaviour. Since each wrapper is executed in its own function scope, the different wrappers cannot access each others' policy state. Due to this fact, the final decision whether to execute an original API `f` is independent of the order in which the policies are injected into the app. In this way, PhoneWrap is completely modular.

The *precision* of the enforced behaviour only depends on the specification of the policy. By describing the granting UI elements with enough properties e.g. their icon, caption, colour or opacity PhoneWrap makes sure that the app cannot generate tickets by making the user interact with harmless looking buttons. In practice, actual end users might rely on third parties to perform this *policy creation* step to provide a policy file which PhoneWrap can insert into the package automatically or a full package with a custom policy injected.

3.1.2 Real-world resource behaviour

PhoneWrap policies also include a few features to better capture the resource behaviour of real-world apps.

First, the policy state contains the additional switches `blockAll` and `allowAll`. They can be set in the starting state of the policy and overwritten by any button policy. If the first switch is set, all access is denied independent of the ticket state. Otherwise, if `allowAll` is set, all access is allowed. The tickets are only subtracted if neither of the switches is set. This way, access to the GPS location in the example app "TrackMyVisit" can be granted unconditionally when the user starts a journey.

Second, other apps, for example, send messages to each contact the user has marked in a list of checkboxes. The consumption is therefore equal to the number of selected checkboxes. In PhoneWrap's button policies, a UI element can be identified as a *checkbox* which instructs PhoneWrap to grant tickets when the box is checked and revoke the tickets when the user unchecks the box.

Finally, some of the real-world examples try to make the user aware of the resource consumption by displaying a *confirmation dialog*. The resource is only accessed if the user presses the confirming button. However, confirmation dialogs are not part of the DOM tree and, therefore, PhoneWrap does not receive events for the dialogs. To incorporate dialogs into policies, PhoneWrap instruments the dialog APIs and wraps the callback functions of each dialog. If a button policy specifies a list of captions in the `confirm` parameter, PhoneWrap, rather than granting the specific number of tickets for this UI event, only reserves them. If in the subsequent dialog a button with one of the specified captions is pressed, the reserved tickets are granted. If the user presses a dialog button not specified in the `confirm` list, PhoneWrap deletes all reserved tickets.

4 Evaluation

The following evaluation was set to answer the following two questions:

1. Does PhoneWrap restrict the resource behaviour to the claimed bounds?
2. Can the expected behaviour of real-world apps be described by PhoneWrap policies?

For the first question we applied PhoneWrap to a specifically crafted app which executes various code snippets to circumvent PhoneWrap's wrapping and access the vibration service. The vibration feature of the phone was chosen as the resource, for its immediate effect. We fitted this app with a policy allowing the vibration only for a control button. PhoneWrap was able to successfully deny access to the resource while the control button preserved its functionality and functionality independent of the vibration was preserved as well.

For the second question we applied PhoneWrap to a set of real-world apps. From the test set of 8757 PhoneGap app packages PhoneWrap can inject a policy into 6843 (78%). The remaining apps either use a very early version of PhoneGap where the package structure was not fixed yet or include the PhoneGap library without using the PhoneGap framework. Most earlier versions of PhoneGap could be injected using PhoneWrap by manually adopting the wrapping script to the version.

We chose the messaging service as test resource since it is used in real apps with verifiable and quantifiable result on the user's privacy and phone bill. In the candidate set we found only 10 apps sending messages through a PhoneGap plugin, which were subjected to a closer examination. The behaviour of these apps was inspected manually to identify the resource behaviour, PhoneWrap was applied to the app to enforce the identified behaviour and the behaviour of the modified app was manually verified. Since PhoneGap does not offer a default messaging plugin, we found 5 different PhoneGap messaging plugins with different APIs[1].

[1] more details on the apps and APIs: `https://github.com/DFranzen/PhoneWrap`

4.1 Results of Real-World Evaluations

We were able to describe the resource behaviour and the required bounds for all 10 apps precisely after a few minutes of interaction with the app. The resulting PhoneWrap policies are summarised in Figure 4. The policy for "TrackMyVisit" (9) is shown in Figure 5, all other policies are available at `https://github.com/DFranzen/PhoneWrap`. It shows that each button with an icon ending in either `images/emergency_icon.png` or `images/emergency_icon2.png` generates 3 local tickets per click when confirmed by the dialog button "Yes". The policy guards the JavaScript API `smsplugin.send`, the Java API `SmsPlugin.SEND_SMS` and the plugin function `cordova/plugin/smssendingplugin.send`.

Apps 1-5 are fitted with a simple policy where a specific button is allowed to send exactly one message. Apps 6-9 display a confirmation dialog before the message is sent which is captured with the `confirm` policy parameter. App 9 sends up-to 3 messages for each button press. Local tickets prevent the app to abuse the unused tickets later. App 10 lets the user select contact numbers from a list and later sends 1 message to each selected contact which is captured by the `checkbox` policy. Here, PhoneWrap needs to create non-local tickets since ticket generation and ticket consumption are triggered by separate events. All other apps (1-8) spend generated tickets immediately which makes local and non-local tickets equivalent. Where possible, local tickets are preferred as the fail-safe behaviour.

In app 8, we were able to improve the resource behaviour by restricting unwanted resource consumption, probably caused by a bug. This app displays a confirmation dialog before the message is sent. However, we discovered that the message is sent regardless, even if the user presses the "Cancel" button. The injected PhoneWrap policy grants the message only if the "OK" button is pressed.

The apps 5 and 7 send charged premium messages. Before such a message is sent, Android warns the user in a confirmation dialog. This dialog is out of the scope of PhoneWrap, since PhoneWrap enforces its policy on the application level. As a consequence, the dialog is displayed if the PhoneWrap policy grants the message and suppressed as part of the API call if the PhoneWrap policy denies the message.

5 Related Work

The wrapping method used in this work is inspired by Self-Protecting JavaScript [15], but extended to mobile apps and the concrete interaction-dependent ticket-based policies. The improvements of Magazinius et al. [13] apply in the same way to PhoneWrap. Access Control Gadgets [18] propose similar interaction-dependent policies. However, rather than augmenting UI elements of the app itself to generate tickets, they require each library to provide special permission granting UI elements which can be embedded into the app. Furthermore, their approach requires a modified execution environment to capture the events and protect the privileged UI elements.

There are several other frameworks to enforce policies for JavaScript. Compared to our approach, they either do not take user input into account [12, 4] or modify parts of the operating system or the browser [16, 14, 8, 17].

Security systems with quantitative policies have mainly been studied for Java [5, 19]. Since Java applications are usually compiled, the light-wight enforcement contained within the language used by PhoneWrap is not possible here. However, similar approaches achieve wrapping by taking advantage of the dynamic linking to libraries [20]. The tool Dr. Android and Mr. Hide [10] implements a finer-grained access-control permission system for Android by wrapping APIs. Like PhoneWrap, it modifies Android apps and repacks them, but only refines the general permissions and does not consider quantitative or interaction-dependent policies.

6 Conclusion and Discussion

We presented the system PhoneWrap which injects interaction-dependent ticked-based policies into mobile apps written in JavaScript by wrapping the resource consuming APIs. The implementation extracts all necessary information from standard Android packages and automatically inserts user-created policies into real-world apps. Therefore, it is usable without knowledge of the code of either the guarded app or of the PhoneWrap system and executes real-world apps on an unmodified mobile phone system.

PhoneWrap can enforce policies on all API activated resources. However, it only enforces a bound on the number of calls to the critical API not the use of potentially returned values. For example, PhoneWrap does not restrict the app from sharing obtained private information. This can be achieved by different techniques like flow analysis.

Using the tool, we successfully identified and injected appropriate policies into 10 apps restricting their behaviour to a fine-grained least-privilege access to the messaging service. A larger scale evaluation has to show whether these examples are representative and whether additional resource behaviour patterns like the `checkbox` and `confirm` pattern need to be covered.

Similar results could be achieved by *rewriting*

	App (version, versionCode)	Button Condition	Tickets	Policy features used
1	com.GPAInsurance.myinsurance.apk(1.0, 2)	Caption: “Send Text”	1	
2	nu.fdp.Boatsteward.apk(1.5, 6)	id: “btnSendTheMessage”	1	
3	nu.fdp.optimaxx_gsm.apk(2.6, 26)	id: “btnSendTheMessage”	1	
4	nu.fdp.Sms_RC.apk(4.1, 19)	Caption: “Send”	1	
5	se.fjellandermedia.tidegarden.apk(3.1, 310)	Caption: “Ge med SMS” id: “donateSMS”	1	
6	nu.fdp.Sms_RC_Mini.apk(1.7, 8)	src: (ends with) “/pict/send.png”	1	confirm: “Send”
7	no.idium.apps.maf.apk(1.0.1, 68)	id: “stage_Gi”	1	confirm: “Ok”
8	no.idium.apps.apk(1.0, 52)	class: “sms_small”	1	confirm: “OK”
9	myzealit.TMV.apk(2.2, 22)	src: (ends with) “emergency_icon.png”	3	confirm: “Yes”, local
10	com.ServiceHours.ServiceHours.apk(1.3.3, 8)	name: “contactnumber”	1	checkbox

Figure 4: Apps and their Policies

```
policy = {
  mperms : 0,
  buttons : [
    { cond: {
        src:"images/emergency_icon.png"
      },
      match:"ends",
      mperms:3,
      local: true,
      confirm: ["Yes"],
    },
    { cond: {
        src:"images/emergency_icon2.png"
      },
      match:"ends",
      mperms:3,
      local: true,
      confirm: ["Yes"],
    }
  ],
  guard: ["smsplugin.send"],
  guard_exec: ["SmsPlugin.SEND_SMS"],
  guard_require: ["cordova/plugin/
      smssendingplugin.send"],
  deny: function(){alert("Policy: Denied")}
}
```

Figure 5: Policy: TrackMyVisit

the JavaScript code, which is more difficult due to JavaScript's dynamic nature and its ability to rewrite itself in the DOM tree. PhoneWrap naturally handles these JavaScript features, since the original methods are protected in the JavaScript scope rather than the DOM tree. Changing the *native part* could also capture all resource consumption of PhoneGap apps. However, the native part is only available in compiled form and the multiple different versions of the PhoneGap library make it difficult to achieve consistent modifications. In comparison, a PhoneWrap policy works for all versions of the PhoneGap library as long as the API of the plugin stays unchanged. Compared to static analysis, PhoneWrap does not over-approximate all possible runs of the app and can even alter the app to adhere to the policy in the case of policy violation. In parallel work [7] we infer the quantitative resource behaviour of unmodified JavaScript code, where possible, resulting in bounds with similar shape. The results of either system could improve the other.

Before PhoneWrap can be deployed in real world scenarios, it needs some improvements. Most importantly, in the current version the policy creator has to identify all critical plugins and APIs manually. Future work should include a relation between resources and corresponding plugins (as provided at `https://github.com/DFranzen/PhoneWrap`) to automatically find the critical APIs and to propose which resources to guard. Extracting also the correct bounds from the description of the app is an interesting and useful addition to this system, but would be subject to a different field of research. Finally, for integrity, PhoneWrap needs to verify the PhoneGap library and the plugin files included in an app to ensure that no hidden APIs have been injected. This can be achieved by checking against a white list of all official versions.

References

[1] Adobe Systems Inc. Adobe phoneGap homepage. `http://phonegap.com/`. Accessed March 2016.

[2] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. v. Styp-Rekowsky. AppGuard Fine-Grained Policy Enforcement for Untrusted Android Applications. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 213–231. Springer Berlin Heidelberg, 2014.

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 274–277. ACM, 2012.

[4] A. Barthe, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[5] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35, 1998.

[6] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999*, pages 32–45, 1999.

[7] D. Franzen and D. Aspinall. Towards an amortized type system for JavaScript. In *6th International Symposium on Symbolic Computation in Software Science (SCSS)*, pages 12–26, 2014.

[8] W. D. Groef, D. Devriese, and F. Piessens. Better Security and Privacy for Web Browsers: A Survey of Techniques, and a New Implementation. In *Formal Aspects of Security and Trust*, pages 21–38. Springer Berlin Heidelberg, 2012.

[9] J. Hakimi. Play Store user review of DTEK by BlackBerry. `play.google.com/store/apps/details?id=com.blackberry.privacydashboard`, Nov. 2015.

[10] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, page 314, 2012.

[11] J. Jung, S. Han, and D. Wetherall. Short Paper: Enhancing Mobile Application Permissions with Runtime Feedback and Constraints. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM*, pages 45–50, 2012.

[12] M. T. Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan. SafeScript: JavaScript Transformation for Policy Enforcement. In *Secure IT Systems*, pages 67–83. Springer Berlin Heidelberg, 2013.

[13] J. Magazinius, P. H. Phung, and D. Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Information Security Technology for Applications*, pages 239–255. Springer Berlin Heidelberg, 2012.

[14] L. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 481–496, 2010.

[15] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*, pages 47–60. ACM, 2009.

[16] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[17] G. Richards, C. Hammer, F. Zappa Nardelli, S. Jagannathan, and J. Vitek. Flexible Access Control for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 305–322, 2013.

[18] 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 (SP)*, pages 224–238, 2012.

[19] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers. Strong Mobility and Fine-Grained Resource Control NOMADS. In *Agent Systems, Mobile Agents, and Applications*, pages 2–15. Springer Berlin Heidelberg, 2000.

[20] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22nd USENIX Security Symposium*, pages 559–572, 2013.