Short paper: Rethinking Permissions for Mobile Web Apps: Barriers and the Road Ahead

Chaitrali Amrutkar and Patrick Traynor
Converging Infrastructure Security (CISEC) Laboratory
Georgia Tech Information Security Center (GTISC)
Georgia Institute of Technology
chaitrali@gatech.edu, traynor@cc.gatech.edu

ABSTRACT

The distinction between mobile applications built for specific platforms and that run in mobile browsers is increasingly being blurred. As HTML5 becomes universally deployed and mobile web apps directly take advantage of device features such as the camera, microphone and geolocation information, this difference will vanish almost entirely. In spite of this increasing similarity, the permission systems protecting mobile device resources for native[1] and web apps are dramatically different. In this position paper, we argue that the increasing indistinguishability between such apps coupled with the dynamic nature of mobile web apps calls for reconsidering the current permission model for mobile web apps. We first discuss factors associated with securing mobile web apps in comparison to traditional apps. We then propose a mechanism that presents a holistic view of the permissions required by a web app and provides a simple, single-stop permission management process. We then briefly discuss issues surrounding the use and deployment of this technique. In so doing, we argue that in the absence of an in-cloud security model for mobile web apps, client side defenses are limited. Our model can provide users with a better chance of making informed security decisions and may also aid researchers in assessing security of mobile web apps.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Mobile devices, web app security, permissions

1. MOTIVATION

The family of HTML5 technologies is set to dramatically change the way in which applications are designed for mobile devices. In particular, HTML5 provides direct support for features including audio, video and geolocation information. While access to these features has become a mainstay of "native" mobile apps, their inclusion in this standard makes it possible for mobile web apps to provide many of the features currently implemented by their native counterparts. Direct support for these features within mobile browsers will be transformative - whereas app makers have traditionally had to invest significant effort to develop software across multiple platforms, HTML5 will allow developers to rely on the mobile browser to deliver a single codebase and a unified user experience.

Mobile web browsers provide web apps with access to features including cookies, Javascript, native code and Flash by default. This may lead to granting more than necessary privileges to certain web apps. For example, a file-sharing app has access to Flash by default but may not need it for proper functioning. Access to the default browser features, potentially sensitive hardware (e.g., camera, microphone) and data (e.g., GPS location, contact information) require protection when provided to web apps. While support for HTML5 features is currently limited in mobile web apps, desktop browsers providing such features typically prompt users on a per-use, per-site basis. Mobile browsers too have adopted the same per-use, per-site permission model and thus, suffer from several weaknesses. The current permission model for mobile web apps does not provide a holistic view into the permissions required by an app. Providing a single interface containing every permission that may be used by an app allows both users and security experts a better opportunity to assess the potential for malicious behavior by the app. As an example, the Android manifest file and install-time warnings have successfully served as the basis of a wide-range of malware-detection tools [9, 10, 13–15, 24, 25]. However, we note that simply extracting the underlying platform's (such as Android) model and mapping web apps' requests directly to the platform APIs [20] is not the best matching since the structure of mobile web apps and native apps is different. We discuss several differences in the factors contributing to the security of native apps and web apps. We then argue that there is a need to define permissions for web apps separately while maintaining as much overlap with native app permissions as possible for user learnability and reducing developer efforts. Finally, we argue that the dynamic nature of mobile web apps necessitates a one-stop, easy-to-use permission model that can allow users to access, modify or change the permissions granted to individual web apps.

In this paper, we present a proof of concept mechanism that addresses some of these weaknesses and allows a subset of the Android application security model to be easily expressed by remotely stored mobile web apps. Our goal is to provide expert users with a single interface that allows them to reason about the permissions requested by mobile web apps similar to native apps. We start with

[1]By native, we mean built for a specific platform (e.g., Android, iOS, etc), not necessarily that they contain compiled native executable libraries or code.

a clean slate with no permissions given to a web app by default. We argue that developers should be required to declare permissions needed for a web app up front including the permissions provided by default in current browsers. We then discuss a hybrid approach of install-time and run-time permission authorization. We propose *webifest*, an XML file similar to the manifest in Android framework that includes permission declarations and can be used to allow mobile web apps to provide a concise declaration of the resources they intend to use. We argue that encouraging mobile web app developers to request fewer privileges will reduce the attack surface.

We are careful to note that our proposed solution is *not* a tacit endorsement of the Android permission model in particular. Specifically, we are not arguing that users fully pay attention to and understand all Android permissions. Should a better model be found, *we would still argue that all mobile applications should be evaluatable through a security interface providing a complete view of potential behavior.* Given its extensibility and the success with which security experts have had using it to detect malicious applications, we simply rely on the Android model to illustrate our point.

The remainder of this work provides a brief overview of our proposed architecture and attempts to consider best practices for webifests.

2. BACKGROUND

We discuss factors affecting security of mobile web apps and compare the significance of the same factors in the security of native apps. Building on this analysis, we discuss the potential requirements in defining a permission model for mobile web apps.

2.1 Security Factors

Web apps provide cross-platform functionality, reduce developer effort and are easy to update. However, these useful properties also make web apps difficult to secure. We discuss several factors that contribute to the security of a web app and compare the security consequences of each of the factors with those of native apps.

- Nature of permissions: Mobile web apps are more dynamic in nature because web application providers can easily update the server side code. This dynamic nature of web apps allows frequent changes to the permissions required for execution. In comparison, it is difficult to silently make such changes to native apps. Current web browsers also do not provide an interface to view all the permissions required by a web app, unlike a native app.

- User effort: One of the primary protections for native apps is the attacker has to lure the user into installing his app. It is much easier for an attacker to lure a user into following a link to the attacker's web app through email or advertisements.

- Application markets: Native apps have some level of security through app markets (such as Apple and Google) that detect and remove malware. No such mechanism exists for mobile web apps. Unless an online marketplace such as the Chrome webstore [3] is formed for the mobile web, having a centralized system for security analysis will not be possible. Moreover, if such a system is developed, the dynamic nature of web apps will require continuous monitoring to ensure security. Therefore, permission/capability detection during application submission to the marketplace (e.g., the MSIL code analysis performed on the Windows Phone Marketplace [6]) may not work well for mobile web apps.

- Identical app logic: It is not possible to ensure that all users of the same web app receive an identical copy of the app unlike native apps. Therefore, web apps can collect contextual data such as location more easily by asking different permissions from different users. Due to all users possesing an identical copy of a native app, user-rating systems in native application markets work well. These ratings allow an average user to decide whether an app is good or bad. No such protection exists for mobile web apps. Moreover, even if a third-party entity such as Google independently discovers that a particular mobile web app is bad, conveying this information to the future visitors of the web app is difficult.

- Default permissions: Android native apps are provided with no default permissions. Web apps running in a browser are provided with permissions to access several browser resources such as cookies (maintaining SOP), Flash, download code to the device and run Javascript. Therefore, certain web apps may end up with more privileges than required for their execution.

This is not a comprehensive list of all the factors associated with the security of web and native apps. Other factors including security of the underlying operating system [16], browser [7,8], identity management using certificates also impact mobile web app security. We have discussed the major factors that lead to differences in the security of web and native apps.

2.2 Reflection

The comparison between native and web apps shows that additional security factors need to be considered while designing permission systems for managing access requests from mobile web apps. However, we also observe that following wide adoption of HTML5, both native and web apps on mobile platforms will request access to similar user sensitive data and hardware. Therefore, we choose an approach that will build upon the permission model for native apps and also provide additional features required by mobile web apps.

We anticipate more requests to sensitive information from a mobile web app as compared to a desktop web app. This is because a mobile device can provide contextual information such as location unlike desktop. Current desktop and mobile browsers request runtime permissions for each feature requested by a web app. Desktop browsers such as Safari allow permissions to be stored for a specific time duration. Other browsers such as Chrome desktop store the location permission given to a website forever unless the user revokes it. However, current browsers do not allow a user to view all the permissions used by a web app nor do they allow a user to easily access, modify or revoke permissions.

The lack of a centralized authority such as an app market increases the probability of malicious web apps on the mobile platform. We argue that mobile web app users should be able to easily revoke or selectively grant permissions without being overwhelmed with warnings and thus suffering from warning fatigue.

3. PROPOSED ARCHITECTURE

Goal: We want to provide expert users with a single interface that allows them to reason about the permissions requested by mobile web apps similar to native apps.

We use the Android system as an example and argue that our central idea can be applied to any mobile platform. Our proposed model provides a user consent permission system that gives full view of the required permissions, alerts the user when a web app asks for dangerous permissions and also allows easy revocation of permissions granted to individual web apps. We propose that a web app developer declares all the required permissions to the browser using an XML file similar to the Manifest file in Android. In addition to specifying *how* to access a particular phone feature, our model requires a web developer to define *what* he wishes to access in the form of permission. For example, when a web developer uses the HTML5 Geolocation API to access location, he should specify that the app would require the corresponding permission ACCESS_COARSE_LOCATION as defined in the Android framework.

Permission categorization: We propose a design where the set of permissions given to a web application by default is null. Therefore, to access resources such as Flash, a web developer will have to request permission. This behavior is significantly different than the behavior of current browsers where a website rendered in the browser runs with full browser-privileges and only requests user permission for access to features such as location. We note that we always allow web apps access to the core platform technologies defined in the browser technologies for HTML5[2] [17] and do not consider them in the permission system. The motivation behind requiring developers to explicitly request access to default browser features such as Flash is to encourage web developers to request least privilege. However, the current per-use, per-site permission model would generate multiple warnings every time a user accesses a web app if universally implemented features such as cookies have to be authorized. This may lead to warning fatigue [4, 12, 19, 22] and careless clickthrough.

We propose classifying permissions required by mobile web apps into two categories, normal and dangerous. We argue that a web app connected to the Internet should not be allowed access to permissions corresponding to the `signatureOrSystem` category [1] in the Android permission system. We also note that permissions in the `signature` category [1] loosely correspond to the already implemented Same Origin Policy in browsers. Out of the 75 normal or dangerous [18] permissions provided by the Android framework, warnings are generated only for the dangerous permissions at install-time. We argue that permissions required by web apps should also be treated in a similar fashion. However, *we note that the set of permissions for web apps falling under the normal and dangerous categories would differ from the ones defined in the Android framework and not all of the permissions for native apps on Android would be relevant to web apps*. For example, permissions such as BROADCAST_SMS and BROADCAST_PACKAGE_REMOVED are not relevant for web apps. Additionally, permissions such as INTERNET are crucial for a web app to work unless it is working in offline mode. Therefore, we envision that the number of permissions that a web app can request would be much lower than 75. Even if the browser produces run-time warnings for time-of-use dangerous permissions, we envision that the number of warnings would be limited.

We propose categorizing permissions based on whether a permission will provide access to a user's private data. Normal permissions would be the ones that are crucial for the basic functionality of a web app. Examples of a normal permission are INTERNET, ACCESS_WEBSTORAGE etc. Examples of dangerous permissions would be CAMERA, CALL_PHONE and RECORD_AUDIO. The normal permissions will be granted by the browser without user consent while a warning will be generated when a web app requests access to a dangerous permission.

Managing webifests: We propose a *webifest*, an XML file that can be used by web developers to define all the permissions required by an app. The browser intercepts all webifest files sent by the website in the top-level address bar. For example, consider the following webifest:

```
<webifest domain=foo.com>
<permission=INTERNET, ACCESS_COOKIES,
CAMERA, ACCESS_COARSE_LOCATION>
</webifest>
```

This webifest indicates that a web app from the domain `.foo.com/` is requesting permissions to access the Internet, cookies, camera and coarse location.

When a web app is loaded, the browser intercepts the webifest file only if it is sent over an HTTPS connection. This is to avoid a man-in-the-middle attacker changing the permissions requested by a website. The browser then searches existing webifests for one that matches the domain of the new webifest. If an old webifest does not exist, the browser parses the permissions in the webifest into the normal and dangerous categories. The browser provides normal permissions to the web app without user consent.

Instead of generating an install-time warning for all the dangerous permissions at once similar to native apps, the browser then uses the run-time warning model. When a web app requires a dangerous permission, the browser generates a warning and the user has to approve the permission. Note that the browser does not store the approval when user consents through a warning message. However, the browser provides an interface for the user to selectively approve, store or revoke permissions by using an interface users are already familiar with. We propose extending the interface provided by browsers to access SSL certificate information to accommodate permissions. For example, consider the Chrome mobile browser shown in Figure 1. Clicking on the lock icon in Chrome mobile opens a dialogue that provides identity information of a website. A Chrome mobile browser running our proposed model will also provide an interface to store or revoke *any* of the permissions requested by a web app. When the browser intercepts a webifest, it generates a list of permissions requested by the web app. This list contains both the normal and dangerous permissions and the status information of whether they have been approved. Once a user 'stores' a permission, the browser retains the consent and the user is not asked to approve permissions on subsequent visits to the same web app until the webifest is either revoked by the user or modified by the corresponding web app.

If a browser finds an older webifest for the same 'domain' as that of a newly received webifest, the browser compares the values of the 'Permissions and Domain' attributes of the old and the new webifest. If the attributes are the same, the browser simply ignores the new webifest. Otherwise, the browser generates a warning for the user about a new webifest sent by the web app and displays the additional permissions required. If the user authorizes the new webifest, the browser stores the new one and discards the old copy. The browser still requires run-time approval of additional permissions if any in the new webifest. If a user rejects the new webifest, the web app continues to execute with the permissions in the old webifest until the user deletes the webifest.

We note that our proposal is only for a website requesting permissions of the underlying system. However, we observe that in addition to requesting necessary permissions, a website can also register web-intents with the browser using the webifest file, similar to the manifest file in native apps.

Storing webifests: We propose that a webifest for each web app should be stored using the HTML5 web storage [23]. Web stor-

[2]The HTML5 core platform technologies are HTML, CSS, DOM and Javascript.

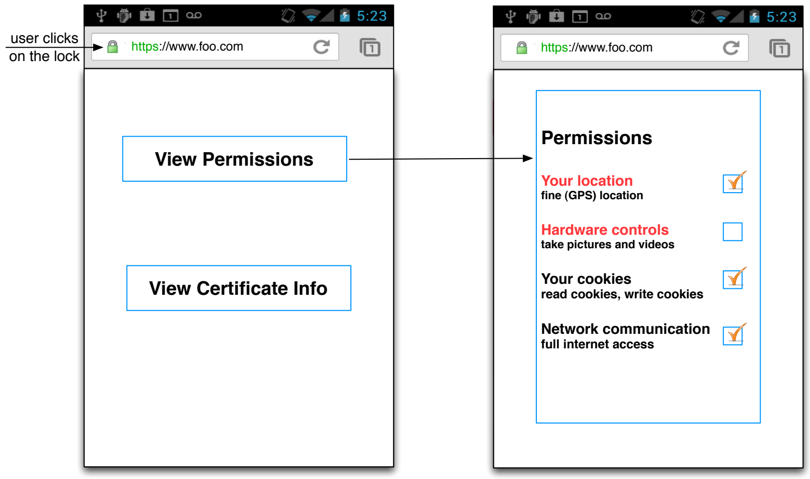

Figure 1: User interface for permission management of mobile web apps. Left: When a user clicks on the lock icon, the browser shows this interface to interact with permissions and certificates. Right: The browser provides an interface to view the status of the permissions requested by `www.foo.com` (domain in the address bar). The user has not stored hardware controls permissions, whereas he has stored the location, cookies and Internet access permissions. The cookie and Internet permissions are normal permissions granted without user consent. The location and hardware control permissions require explicit user consent. The user can easily revoke a permission by unchecking the box next to it.

age supports 'local storage' which is similar to persistent cookies. When a browser intercepts a webifest, the webifest file is stored in the local storage allocated to the corresponding web app's domain. The browser also maintains the status of user approval on all the permissions. Javascript is not allowed access to a webifest unlike other local storage objects.

Managing access requests: When a web app requests access to a phone feature such as camera, the browser searches for the corresponding webifest. The browser verifies whether the user has already granted the requested permission and if granted, allows access without user intervention. If a webifest corresponding to a web app does not exist or the user did not store consent to access the resource requested by the web app, the browser does not allow access. If the permissions required by a web app change, the web app is expected to send a new webifest with the modified permissions in the HTTP response header.

Revocation: Revoking permissions from individual web apps is straightforward. To revoke all the permissions, a user can simply delete the corresponding webifest from the local storage of an app. Alternatively, the user's browser is required to provide an interface that allows the user to revoke all permissions using the in-browser interface. A user can also selectively revoke permissions granted to individual web apps using the in-browser interface shown in Figure 1. Current mobile browsers do not provide an interface to clear access granted to individual websites. For example, in the Android, Dolphin, Firefox Mobile and Opera Mini browsers, a user is required to revoke location access from all web apps at once. This browser behavior precludes a user from revoking permissions granted to only one app if he has provided location authorization to multiple web apps such as Yelp and Google maps. Existing browsers will need modifications to support the proposed revocation procedure.

Conditions: We require that a browser supporting our *initial* model allow only the domain displayed in the address bar of a web app (top-level domain) to save a webifest. Secondly, a cross-domain element embedded in the web app is prohibited from requesting access to hardware or data.

The reasons for allowing only the top-level domain of a web app to save a webifest are the following: a web app may contain cross-domain embedded elements such as advertisements in iframes. If a browser processes the webifest received in the HTTP response header of such an element, there are two ways of obtaining user authorization for the permissions requested by the embedded element. The browser can create separate authorization warnings for the webifest received from each domain or the browser can combine the permissions requested by all domains and create one warning listing all the permissions and the respective domains. The former can overwhelm the user and latter may provide a false sense of security. In the latter scenario, a user may perceive the set of permissions listed in the long warning as the permissions required by the top-level web app to execute. Prohibiting embedded elements from saving webifests can address these issues. Therefore, a browser supporting our model ignores webifests received from embedded elements in a web app.

The second condition of disallowing a cross-domain embedded element to request access to hardware or data follows as a consequence of enforcing the first condition. Since our model requires a webifest for every 'domain' to be able to access a resource, if the browser ignores webifests received from embedded elements, the corresponding domains are unable to access resources.

Consider a web app at `foo.com` that saves a webifest in the browser when the user visits the web app. Later, if an iframe from `foo.com` is embedded in another web app accessed by the user, `foo.com` may try to access resources using the already existing webifest. To avoid this, we require that the browser ignore access requests made by cross-domain elements embedded in the top-level web app. Cross-domain embedded elements only have access to the Internet. We explore the consequences of these conditions in the discussion section.

Size and content: The expected memory overhead in integrating the proposed model in current browsers would be insubstantial due to the small size of webifest files.

Current web apps in mobile browsers do not require all the information provided in the manifest file for native apps for their execution. In addition to declaring required permissions, the manifest file in a native app describes application components such as activities and broadcast receivers, declares permissions required by other apps in order to access the app, the libraries that the app is linked against and the minimum level of the Android API required by the app [2]. Most of this additional information is required since a native app code resides on the device and is always available. This is not true for web apps. For example, at present there is no mechanism that enables a web app to handle an intent created by a native app or the Android system. Whether such a mechanism is possible is an interesting project beyond the scope of this work. Since the compelling need for current mobile web apps is effectively handling permissions, we chose a lightweight manifest file.

4. DISCUSSION

The goal of the proposed model is not necessarily to ensure that average users make better security decisions, but instead to provide an interface for expert users to be able to assess the potential behavior of web apps. If a security expert is unable to evaluate whether a web app is malicious, how can an average user be expected to do the same? Nevertheless, we believe that maintaining significant overlap between the permission models for native and web apps will help average users in making informed security decisions. More importantly, security experts will be able to use the proposed model to design tools similar to the malware detection tools for native apps [9, 10, 13–15, 24, 25].

There are other efforts in the area of permission management for mobile web apps such as Mozilla's WebAPI [5]. However,WebAPI aims at providing consistent APIs that will work in all web browsers and our motivation is designing a permission model for mobile web apps that increases security and user control. Another related effort in restricting the capabilities of web apps is the Content Security Policy (CSP) [21]. CSP enables the authors of a web app restrict from where the application can load resources, whereas our proposal deals with restricting web apps' access to the resources on a user's device. We discuss the advantages and disadvantages of the proposed model.

Pros: Although the normal permissions are granted without user intervention, their use will enable developers to request least privilege. For example, the default Android mobile and Opera Mini browsers support Flash and videos can be played easily in the browser without user consent. However, a word to PDF converter web app may not require Flash support. Requiring a web developer to ask for Flash permission explicitly may reduce overprivileged web apps.

The simple update and revocation procedure for permissions granted to web apps maintains the highly dynamic nature of web apps while allowing the user to revoke permissions easily at will. For example, if a security expert wants to disable cookies for a suspicious looking webpage, he can do so using the selective permission revocation model (desktop browsers have a similar functionality for cookies). The user will not have to disable cookies across all webapps in this case, a provision currently available in mobile web browsers. An average user on the other hand will not have to worry about basic details of browser management due to the normal permissions being provided by the browser without user consent.

Providing a similar user interface for permission management as that of the native apps facilitates user learnability of the proposed web app permission model. Moreover, striving for maximum overlap between the permissions defined for native and web apps may reduce the effort required to secure mobile apps (native and web) and also reduce developer effort.

Cons: Prohibiting a cross-domain embedded element such as iframe from storing webifests may break the logic of certain mobile web apps. However, we argue that due to the constrained nature of mobile browsers, the complexity of mobile web apps is lower. Therefore, the number of cross-domain embedded elements in mobile web apps is expected to be minimal. We plan to investigate models allowing a cross-domain embedded element to access resources. One such method would be to fall back to the per-use, per-website model currently used in browsers. For example, if a non-Google web app includes a Google map, the browser can generate an authorization request for location access when a user wants to interact with the map and never provide the interface to store the permission. Another potential technique would be to fetch a webifest file when a user interacts with a cross-domain embedded element. For example, if a non-Google web app includes a Google map, when a user wishes to interact with the map by clicking on it, the browser fetches the webifest file for the map and generates an authorization request to access the resources required by Google maps, again with no storing facility.

Our model mandates explicit authorization for cookies and ignores webifests sent by embedded elements. This prevents third-party cookies in a webpage from tracking a user across sessions. However, due to the universal usage of cookies, not allowing cookies from other domain excluding the top-level domain in the address bar may break several websites. Whether our model should allow cookies for embedded elements is a topic for future work.

If a web application does not support webifests, a browser supporting webifests can default to the current model of processing permissions based on per-use, per-website. Although this approach maintains backward compatibility, it defeats the purpose of our model. We assert that a browser should support the same permission model for all the elements of a web app irrespective of their domain. If the top-level web app supports webifests, the browser should use the webifest model for all the elements. Otherwise, the browser should use the per-use, per-website model.

Adopting the proposed model will require collaborative efforts from web developers and browser vendors. Finally, storing the webifests file may be tricky since implementing local storage features in a browser may pose information leakage or information spoofing risks [23].

Security comparison with native apps: The notion of security in the proposed model is similar to that of the permissions model for native apps. The model does not protect users against malicious apps that can access sensitive data as a result of user authorization. The level of security depends on a user's knowledge about permissions and the consequences of authorizing a web app to access sensitive information. Moreover, the model cannot provide the guarantee that a web application would request the least privileges required for its execution. Researchers have previously shown that native app developers attempt to obtain least privilege for their applications, but fall short due to API documentation errors and lack of developer understanding [14]. We expect the same effort from mobile web app developers. We also note that developers that change the permissions required by their app too often might turn away users. Since the browser does not store a permission unless explicitly approved by the user and also alerts the user when a new webifest is detected, multiple changes in webifests may make the user suspicious or annoyed.

Research questions: Several research questions present themselves:

- Normal versus dangerous permissions, where to draw the line? We based our initial proposal of categorization on whether a web app requests permissions to access a user's private data. Are there other possible categorization procedures that can further reduce warnings?
- If the set of normal permissions is large, would it be problematic to grant normal permissions without user consent?
- Would users and developers understand the new model easily?
- Can a mechanism other than local storage be used to maintain webifests? Using local storage requires creating a special exception for preventing Javascript from accessing webifests.
- Is warning fatigue possible for the proposed model? If the total number of permissions available for web apps on a platform is limited, the probability of warning fatigue would be curtailed.

Looking forward: Due to the availability of contextual information, we envision widespread use of HTML5 to access user's data on a mobile device. We have taken the first step towards rethinking the permission system for mobile web apps. An alternative permission model would be adopting the install-time all-or-nothing model defined in Android native apps. Yet another client-side solution for the permission management problem would be defining in-browser policies to limit a user's access to potentially malicious websites and also block a website's access to sensitive features such as contact lists. Although this approach is possible for corporate phones, it would be difficult to implement on personal phones of average users. Moreover, restricting functionality can result in poor user experience. To avoid pushing security decisions to the user, multiple security tools such as Zozzle [11] can be executed in the browser to protect the user from malicious web apps. However, due to the hardware limitations of mobile devices, this approach will entail severe performance penalties and will not be practical. Finally, if a mobile web app store similar to the Chrome webstore is available in the future, providing centralized security measures would be possible.

Our proposed model provides more control to the user, but also redirects more security decisions to the user. Offloading security decisions to a user is not the best idea. However, we imagine that in the absence of a proxy like setting or an umbrella webstore, client side defenses on mobile devices would be limited.

Acknowledgments

We would like to thank Italo Dacosta for his insightful feedback. This work was supported in part by the US National Science Foundation (CNS-0916047, CAREER CNS-0952959). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

5. REFERENCES

[1] Android permission categories. `http://developer.android.com/guide/topics/manifest/permission-element.html`.
[2] The AndroidManifest.xml File. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.
[3] The Chrome Webstore. `https://chrome.google.com/webstore/category/home`.
[4] W3C: Web Security Context: User Interface Guidelines. `http://www.w3.org/TR/wsc-ui/`, August 2010.
[5] Mozilla WebAPI. `https://wiki.mozilla.org/WebAPI`, July 2012.
[6] How To Determine Application Capabilities. `http://msdn.microsoft.com/en-us/library/gg180730(v=vs.92).aspx`, April 2012.
[7] C. Amrutkar, K. Singh, and P. Traynor. On the Disparity of Display Security in Mobile and Traditional Web Browsers. Technical report, GT-CS-11-02, Georgia Institute of Technology, 2011.
[8] C. Amrutkar, P. C. van Oorschot, and P. Traynor. Measuring SSL Indicators on Mobile Browsers: Extended Life, or End of the Road? In *Proceedings of the Information Security Conference (ISC)*, 2012.
[9] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. Technical Report GT-CS-12-01, College of Computing, Georgia Institute of Technology, 2012.
[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.
[11] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX conference on Security*, 2011.
[12] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the 26th annual SIGCHI conference on Human factors in computing systems*, 2008.
[13] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.
[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.
[15] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, 2011.
[16] M. Grace, Z. Zhou, Yajin Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.
[17] HTML5 contributors. The Web platform: Browser technologies. `http://platform.html5.org/`.
[18] S. Lekies and M. Johns. Lightweight integrity protection for web storage-driven content caching. In *In IEEE Oakland Web 2.0 Security and Privacy (W2SP)*, 2012.
[19] Security News Portal. Symbian – Just Because it's Signed Doesn't Mean it Isn't Spying on You. `http://www.securitynewsportal.com/securitynews/article.php?TITLE=Just_because_its_Signed_doesnt_mean_it_isnt_spying_on_you`, 2007.
[20] K. Singh. Can Mobile learn from the Web? In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2012.
[21] B. Sterne and A. Barth. Content Security Policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.
[22] D. Stewart and I. Martin. Intended and Unintended Consequences of Warning Messages: A Review and Synthesis of Empirical Research. 1994.
[23] W3C. Web storage. `http://dev.w3.org/html5/webstorage`.
[24] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, 2012.
[25] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.