Secure Integration of Web Content and Applications on Commodity Mobile Operating Systems

Drew Davidson
University of Wisconsin

Yaohui Chen
Stony Brook University

Franklin George
Stony Brook University

Long Lu
Stony Brook University

Somesh Jha
University of Wisconsin

ABSTRACT

A majority of today's mobile apps integrate web content of various kinds. Unfortunately, the interactions between app code and web content expose new attack vectors: a malicious app can subvert its embedded web content to steal user secrets; on the other hand, malicious web content can use the privileges of its embedding app to exfiltrate sensitive information such as the user's location and contacts.

In this paper, we discuss security weaknesses of the interface between app code and web content through attacks, then introduce defenses that can be deployed without modifying the OS. Our defenses feature WIREFRAME, a service that securely embeds and renders external web content in Android apps, and in turn, prevents attacks between embedded web and host apps. WIREFRAME fully mediates the interface between app code and embedded web content. Unlike the existing web-embedding mechanisms, WIREFRAME allows both apps and embedded web content to define simple access policies to protect their own resources. These policies recognize fine-grained security principals, such as origins, and control all interactions between apps and the web. We also introduce WIRE (Web Isolation Rewriting Engine), an offline app rewriting tool that allows app users to inject WIREFRAME protections into existing apps. Our evaluation, based on 7166 popular apps and 20 specially selected apps, shows these techniques work on complex apps and incur acceptable end-to-end performance overhead.

1. INTRODUCTION

A common app design paradigm is to embed web content directly in an app's UI. Apps that follow this paradigm, which we call *web-embedding apps*, combine the advantages of both the mobile web and native apps: web content is highly portable across platforms, and native app code can leverage the full power of the device. Unfortunately, these apps also introduce unique attack vectors in the interactions between web content and app code.

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ACM 978-1-4503-4944-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3052973.3052998

All major mobile platforms offer web-embedding support. The `WebView` class in Android and `UIWebView/WKWebView` class in iOS are UI widgets that display remote web elements or entire web pages natively within an app.[1] Web content and the embedding app can programmatically manipulate each other's data and behavior via the so-called *app-web bridge* APIs. For instance, an app can programmatically configure embedded WebViews and inject scripts. Conversely, JavaScript loaded in an WebView may call exported app code to access local resources, such as the file system, the camera, or GPS.

The popularity of web-embedding apps makes the app-web bridge an attractive target for attacks *from both sides*: a malicious app may seek to subvert or leak sensitive web content (i.e., *app-to-web attacks*); malicious web content may attempt to misuse the app's permissions and local resources (i.e., *web-to-app attacks*). Both types of attacks are increasingly observed in reality [27, 21].

Malicious apps can embed and manipulate web content from sensitive domains. Well-established web security policies, such as the *same-origin policy* (SOP), are not enforced upon app-web interactions, largely due to the simplistic security design of WebView, which presumes apps always own embedded web content. As a result, web-embedding apps can easily disturb or spy on third-party web services, such as single sign-on (SSO) and in-app payment. Furthermore, apps can undermine inter-frame sandboxing by retrieving scripts from one page and injecting them into another. This means that a malicious app is not restricted by the SOP and can introspect on sensitive, third-party web content.

Conversely, malicious web content embedded in benign apps can abuse the app's resources. The permissions granted to an app are implicitly inherited by its embedded web content: the privileges meant for a trusted domain are universally available to sub-frames or elements loaded from untrusted domains in the WebView, allowing malicious web content from one domain to leverage permissions intended for a different domain. Moreover, the app-web bridge allows app developers to make portions of their app code invocable by JavaScript loaded in WebViews. This feature greatly facilitates web content's access to local data and resources such as the GPS location of the device. Unfortunately, this access is not restricted to a given origin. Therefore, developers are often forced to ignore attacks, such as those reported in [16, 27], in favor of adding app functionalities.

[1] Though we focus on the security of Android WebViews, we believe that our observations and techniques are applicable to iOS.

Despite their variety, the above attacks in both directions share a common root cause: Web content providers and app developers have distinct security requirements that current web-embedding mechanisms are incapable of distinguishing or enforcing. For instance, app developers have no means of controlling which web origins can use what app data and code via the app-web bridge—they can only choose to fully expose app interfaces to WebView or not at all (and thereby give up app features). Similarly, web service providers cannot express their needs for isolating their sensitive web content from apps or allow only limited access, and often have to sacrifice security and privacy for mobile integration.

In this paper, we introduce a novel approach to enabling fine-grained, policy-driven security for the app-web bridge. Our approach is trustworthy to both apps and web content providers. It protects both sides from attacks launched by the other side. It is applicable, without OS changes or device rooting, to both current and previous generations app-web bridges. The contributions of our work are as follow:

- We formulate a system of *dynamic access policies* that allows both apps and web content to protect themselves from each other while maintaining the benefits of integrating apps and the web. We provide complete mediation between apps and their embedded web content. We create a technique called *origin tagging* to establish articulated security principals for app-web interactions.
- We implement a static/dynamic hybrid system to deploy our protection mechanisms without modifying the operating system or requiring the cooperation of developers. Our evaluation using 7166 popular apps shows that this system is compatible with existing apps and effective in enhancing the security of web-embedding apps while incurring minimal overhead.

The two components of our hybrid system are (1) a runtime component built as a regular stand-alone app, called WIREframe, serving as a trustworthy provider of secure, isolated WebViews, and (2) a static, offline app rewriting component called WIRE (Web Isolation Rewriting Engine). Web-embedding apps use WIREframe to render their embedded web content in decoupled, mediated WebView instances. WIREframe allows app developers, app users, and web content providers to define their own dynamic access policies, which protect their respective resources. WIREframe's policy enforcement recognizes fine-grained security principals (e.g., web origins and app identities) and controls all app-web interactions. WIRE automates the adoption of WIREframe in existing apps by statically rewriting an app before installation. Each WebView in the app is replaced by a mediated WebView instance in WIREframe. In addition to separating the app from its WebView, this also separates the individual WebViews in the same app.

Previous works have proposed mitigations to several related attacks. They isolate malicious ads in apps [10, 20, 26, 33] or protect embedded web logins [5, 15, 25]. Although effective against their focused attacks, these solutions cannot be generalized to defeat other types of attacks between embedded web and apps. A recent work [29] retrofits origin-based security to WebViews. Although a significant step towards securing the app-web bridge, it concerns only web-to-app attacks and requires deployment support from OS or device vendors. Compared with these works, our solution generally prevents attacks on both directions of the app-web bridge, requires no OS modification or replacement of WebView, and enables fine-grained and policy-driven security trusted by both web providers and app developers.

The rest of our paper is organized as follows: In § 2, we introduce our threat model, discuss the security limitations of current WebView, and present example attacks. In § 3, we outline the designs of WIREframe and WIRE, followed by their technical details in § 4. We provide a security analysis of our system in § 6 and evaluate our prototype implementation of WIREframe and WIRE in § 7. We compare our system with related work in § 8, and conclude in § 10. The interested reader may find algorithmic details and future work in Appendix A and B, respectively.

2. EXAMPLE ATTACKS AND ANALYSIS

2.1 Threat Model

Our system adopts a threat model that considers two separate classes of attacks exploiting the current WebView design:

App-to-Web Attacks: an app may spy on or manipulate embedded web content sourced from a third-party provider. In this case, the app, which controls the WebView, is the attacker; the embedded web content (and its provider) is the victim. To perform the attack, the malicious app may use the WebView inspection APIs or directly manipulate the WebView's data in memory. Moreover, the malicious app may employ obfuscation techniques, including reflection and native code, to obscure its (ab)use of the WebView.

Web-to-App Attacks: an embedded web page from a third-party may attack its host app. In contrast to the previous class of attacks, the content embedded in a WebView (and its provider) is the attacker; the app that hosts the WebView is the victim. In such attacks, the malicious web content may exploit any web-facing interfaces exposed by the WebView and the host app, including the exported Java methods. However, the malicious web content is not expected to exploit arbitrary code execution vulnerabilities in the WebView. These vulnerabilities are rare and out of the scope of this work, which addresses the insecure design of WebView rather than implementation bugs.

We note that attacks in which an adversary controls both web content and app code simultaneously are out of the scope of this work.

2.2 Attack Scenarios

To illustrate the types of attacks that fall under our threat model, we introduce three representative examples. We use these examples to discuss the security limitations of WebView and the app-web bridge that enable the exploits.

2.2.1 SSO Credential Stealing

As one instance of an app-to-web attack, we implemented a malicious web-embedding RSS reader app, WebRSS. RSS readers are widely used on Android, with popular apps such as Feedly and Flipboard boasting hundreds of thousands of installs. WebRSS requires no permissions besides the `INTERNET` permission, which allows the app to access the network and is necessary for a legitimate RSS reader.

Like many account-based apps, WebRSS allows users to authenticate themselves using a third-party SSO service. SSO allows users to forgo the creation of a separate user-

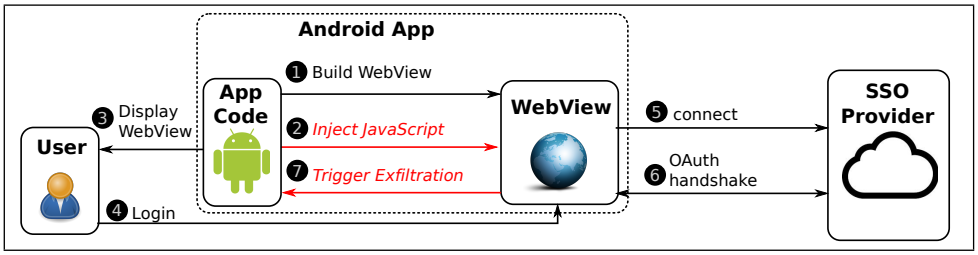

Figure 1: Workflow of an attack on an SSO client, as represented by the example app WebRSS. The app waits for the SSO dialog to appear in the WebView, then scrapes the username and password from the WebView via introspection, either through reflection or injected JavaScript.

name and password combination for each account that they maintain. SSO services are popular precisely because they identify users without directly exposing secret credentials. Instead, users authenticate (by entering a username and password) to a dialog (inside a WebView) controlled by the SSO provider. Upon a successful login, the service passes an opaque authentication token back to the app, which attests to the user's identity without revealing credentials.

However, a malicious app like WebRSS can indirectly obtain user credentials by injecting JavaScript into the authentication WebView to scrape the username and password from the text fields, even when the password field is blinded. Figure 1 illustrates the workflow of this attack at a high level. WebRSS goes through three steps in the attack (relevant snippets of code from WebRSS are shown in Figure 2):

Construct WebView: The first step, shown in Figure 2(a), builds a WebView to load the authentication dialog. Note that the app code enables JavaScript on the WebView and interacts with a real SSO library, in this case LinkedIn. From the perspective of the library, no malicious behavior occurs as the app code is allowed to call `getRequestToken()` to get the opaque SSO token.

Attach JavaScript Bridge: Figure 2(b) shows the app code that will exfiltrate the user credentials. For the purpose of demonstration, this code outputs the username and password to a log file, but could send the values to an adversary over the internet using the permissions already granted to the app for legitimate RSS functionality.

Inject JavaScript Code: To complete the attack, the malicious app registers for a callback when the authentication dialog is loaded, as shown in Figure 2(c). When the callback is fired, the app injects the JavaScript code on Lines 9-15, which is stored as a string as part of the app. The script scrapes the credentials from the dialog and passes it to the code of Figure 2(b) through the app-web bridge. The JavaScript can extract the contents of the password field (Line 12) even though it is blinded (i.e. it displays a series of dots on-screen rather than the literal characters of user input). The characters of the username and password are exfiltrated when the user clicks the "Allow Access" button.

The use of a WebView in WebRSS also enables a web-to-app attack. Consider an iframe containing third-party content (e.g., an ad banner outside of the SSO provider's domain) on the user login page or the redirection page following a successful login. Although the same-origin policy prevents the third-party website from viewing web data from the SSO provider's domain, the third-party iframe can invoke, without restrictions, the Java interfaces exported by the local app and the SSO library. This includes sensitive interfaces solely intended for the web login (e.g., for retrieving user location or login history data). Without breaking any web or app security policy, the malicious iframe can read sensitive data intended for the first-party site using the app-web bridge.

2.2.2 *Local Storage Inference*

A powerful web-to-app attack involves web content loaded in WebView stealing content from the host app. Most recently, Son *et al* observed several such attacks, including instances where web content infers the existence of local files and may read the contents of such files [27]. Such attacks have a severe privacy impact. Son *et al* found cases in which the host app contained information on the user's medications, dating gender preference, social circle, and identity. The host app may contain credentials used to authenticate the user, allowing malicious web content to breach the user's security. The attack relies on specific configuration of the WebView. However, Son *et al* found that such configuration is required and used in legitimate circumstances. Unfortunately, the current design of WebView and the app-web bridge cannot allow apps to selectively expose local resources to web content based its origins or trust levels. When an app needs to permit any trusted web content to access local files or other resources, the same level of access is given to all web content despite their origins.

2.2.3 *User Impersonation*

Another abuse of the app-web bridge is for a malicious app to trick an embedded WebView and impersonate a user through JavaScript actions. Websites are largely defenseless against such actions: even if they require users to manually input credentials and prevent malicious credential stealing (e.g., through a use of a properly salted and encrypted password with every login), a malicious app can simply wait for the credentials to be input and then send surreptitious requests to the authenticated page in the guise of the user.

Such attacks are not just realistic but likely. For instance, attackers often repackage popular websites' official companion apps, which are usually thin wrappers around WebViews. The rogue companion apps can stealthily impersonate users, which is difficult for web servers or average users to detect. Furthermore, apps that allow for general-purpose browsing can include specific triggers on particular websites to launch user impersonation attacks.

2.3 Exploit Analysis

The common cause of the above attacks lies in two assumptions implicit to the design of WebViews: (1) apps own

```
1 public void setWebView(){
2   WebView v = (WebView)findViewById(R.id.w);
3   v.getSettings().setJavaScriptEnabled(true);
4   v.setWebViewClient(new WebClient());
5   v.addJavascriptInterface(new JS(), "js");
6   LinkedInRequestToken t = getRequestToken();
7   v.loadUrl(t.getAuthorizationUrl()); }
```

(a) WebView Configuration Code

```
1 public class JS{
2     void harvest(String name, String pass){
3       Log.e("NAME", name);
4       Log.e("PASS", pass);
5     }}
```

(b) Exfiltration Callback Code

```
1 public class WebClient extends WebViewClient{
2
3  public void onLoadResource(WebView v, String url){
4    super.onLoadResource(v, url);
5
6    String tgtURL = "linkedin.com/uas/oauth/";
7    if (url.contains(tgtURL)){
8      v.loadUrl("javascript:function hack(){"
9      + "var f = document.getElementById("
10     + "'session_key-oauthAuthorizeForm');"
11     + "var g = document.getElementById("
12     + "'session_password-oauthAuthorizeForm');"
13     + "js.harvest(f.value, g.value);};"
14     + "document.getElementById("
15     + "'Allow Access').onclick=hack()");
16  }}}
```

(c) JavaScript Interface

Figure 2: Code snippets from WebRSS to steal user credentials from an SSO dialog. (a) shows the configuration of the WebView from malicious app code, enabling the JavaScript code to be injected and run. (b) shows the app code which is called to exfiltrate user data scraped from the authentication dialog. (c) shows how JavaScript is constructed from within the app and injected into the authentication site.

web content embedded in them; (2) WebView content comes from a single origin. Android provides only a weak form of isolation between the app and web content: the app loads the web content, and can cede coarse-grained control. Since both run in the same process, the app is expected to protect the user from malicious web content. Unfortunately, the weak isolation between apps and web content is insufficient to prevent attacks between apps and embedded web. In the next section, we show how our system improves upon this isolation while still allowing sharing when appropriate.

3. SYSTEM OVERVIEW

In this section, we describe our system and show how it addresses the threats of §2. The key capability of the system is that it provides a secure service that runs web content in a decoupled app. The most obvious benefit of this approach is that it places app and process boundaries between the web content and embedding app, leveraging existing isolation mechanisms without modifying the underlying OS or framework. However, the true power of our approach is that it provides an opportunity for both the app and web content to express dynamic access policies over their interactions. The secure service mediates all interactions between app code and web content over an inter-process communication (IPC) interface subject to these policies.

3.1 System Design

As described in §1, our system consists of two components: (1) a standalone Android app, WIREframe, that runs the secure WebView service. (2) a static, offline rewriting tool, WIRE, that retargets apps to use the WIREframe service. WIRE injects the protection mechanisms of WIREframe without requiring apps to be redesigned. Thus, it ensures that the policies of each security principal are enforced. Next, we describe the operation of our system by walking through its workflow (Figure 3).

WIREframe App: At runtime, WIREframe registers a background service that waits for connections from client apps (i.e., third-party apps using WIREframe). When a connection is created, the service binds a new *IPC Agent* to the client app and establishes a stateful connection via Android's Binder mechanism. If the client app is allowed to display WebViews, the IPC Agent constructs a floating window that contains an actual WebView instance, called the *Concrete WebView*. The IPC Agent maintains an internal mapping between each WebView instance rendered by WIREframe and its counterpart in the client app. Throughout the lifecycle of the WebView, the IPC Agent handles the client app's requests for WebView functionalities. For a given request, it first queries the *Policy Checker*, which serves as a security oracle. The Checker has a default configuration, but can also load policies from the client app side (i.e. defined by developers or app users) as well as policies from the web side (i.e. defined by the web content provider). If allow by the policies, the IPC Agent, invokes the corresponding WebView API. The IPC Agent also forwards invocation results or callbacks back to the client app.

WIREframe places mediated WebViews in individual `Service` components running in isolated processes [2], and therefore, strictly separates them from each other and the embedding app. Process separation prevents reflection, memory mapping and other means of stealthy cross-origin memory introspection. This separation applies to not only WebViews' executions, but also their access to local storage, including the cookie database and the accessible paths in the file system, which prevents WebViews housed in WIREframe, often from different apps, from influencing each other.

In-app WebView Proxy: The *WebView Proxy*, loaded inside the client app, initiates and maintains the connection to the IPC Agent. It also handles client-side data marshalling and unmarshalling. In order to maintain a correspondence to the look and feel of an embedded WebView, the Proxy builds an empty view component (called the proxy view) in the client app and registers callbacks to visual changes to the proxy view. Whenever these callbacks fire, the Proxy forwards them to the WIREframe app to propagate the corresponding view change in the concrete WebView. The proxy maintains the same syntactic interface as an Android WebView. For example, the typical way that a page is loaded in a WebView is by invoking the `loadUrl` method. Thus, the WebView proxy exposes a `loadUrl` method, which it translates into IPC, ultimately resulting in a concrete call to the Concrete WebView within the WIREframe service. We discuss technical details of how this interaction works in §4.

WIRE Tool: Although app developers can interface with the WebView Proxy manually, our threat model assumes

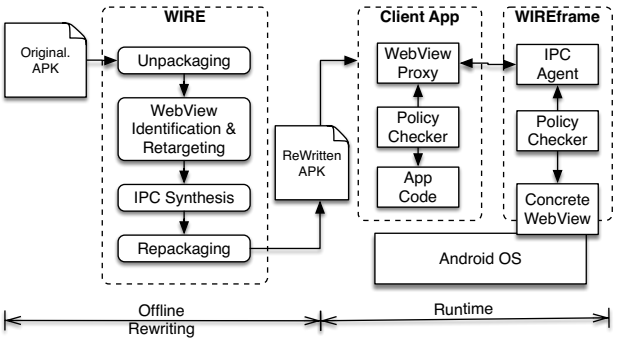

Figure 3: System diagram of WIRE and WIREFRAME. WIRE is applied to a third-party app before install time, ensuring that it uses the protection mechanisms of WIREFRAME at runtime.

that developers can be malicious. As such, WIRE is needed to help app users and IT administrators automatically retarget WebViews in (untrusted) apps into proxy connections to WIREFRAME. WIRE unpackages a given Android APK, and identifies all uses of WebViews. If any such WebViews exist, WIRE injects the WebView Proxy library and replaces all instances of WebViews with instances of the WebView Proxy. This process is aided by the fact that the Proxy has the same interface as the generic WebView. Finally, the app is repackaged, and can be installed on a device, where it will use WIREFRAME. We discuss the implementation of WIRE in §A.

3.2 Dynamic Access Policies

As mentioned above, WIREFRAME enforces access policies to protect web content and app code from one another. By virtue of running each WebView in an isolated process, a web-embedding app can defeat many of the attacks listed in §2: web content can no longer read files from the host app, thereby mitigating local storage inference. The app is disallowed from injecting JavaScript into the WebView, preventing SSO credential stealing and user impersonation.

In the remainder of this section, we discuss additional details on the policy mechanisms and introduce how these policies can be refined dynamically for fine-grained control by each side within a web-embedding app.

3.2.1 Web Protections

The effect of WIREFRAME is to extend the SOP to treat the app code as a distinct origin. A web-embedding app can launch a WebView, but cannot inspect its content. Furthermore, the app is completely disallowed from injecting JavaScript in the WebView. This policy is safe, but it can limit the capabilities of web-embedding apps. For instance, a common behavior of web-embedding apps is to source web content from a remote origin belonging to the app developer, which should be considered as a single origin.

To support this use case, WIREFRAME allows web content owners to declare exceptions via a dynamic policy update mechanism. When the WIREFRAME connects to a remote website, it makes a request for a special set of WIREFRAME specific headers. If the headers are absent, the default policy is employed. If the headers exist, they contain a list of policy objects $\langle A_1, A_2, \ldots, A_n \rangle$. Each policy object A_i specifies a pair (S_i, P_i) where S_i is a security principal and P_i is a policy to enforce over S_i. In our implementation of WIREFRAME, the security principal S_i is an app, identified by its unique app signature and developer's certificate. WIREFRAME verifies the principal identity using the existing signature checking mechanism provided by the OS. A website can also use the `ANY` principal as S_i, which will apply P_i to all embedding apps. The policy P_i is a set of WebView APIs that S_i is allowed to access. For example, if $P_i = \{$ `setJavascriptEnabled` $\}$, then S_i is allowed to inject JavaScript. There is also a special `LOCKDOWN` policy object which puts the WebView into a high-security mode: JavaScript injection is disabled for the remainder of the session.

WIREFRAME and its dynamic policy update mechanism allows web providers to protect their sensitive content or services that are embedded in untrusted apps. For instance, by defining a simple policy that restricts embedding apps' control over the WebViews, web content providers can easily prevent the currently unstoppable app-to-web attacks discussed in §2.2. Note that more complicated policies or more granular principals could be enforced by WIREFRAME (e.g. a policy automaton to prohibit certain sequences of API calls), but our current implementation is sufficient for common use cases. Note that policies are reloaded per-page. Thus, if the user navigates to a new page, policies for previous pages are no longer regarded.

3.2.2 App Protections

A key enhancement that WIREFRAME uses to protect apps from malicious web content (e.g., remote JavaScript calling an exported local Java method) is to regulate requests to the client app on a per web-origin basis. Note that identifying the web origin of a remote request for local resources is not trivial because current WebView design does not provide such information explicitly via its APIs. We obtain the origin information without modifying WebView using a technique called *origin tagging*. By using existing WebView callback interfaces, WIREFRAME rewrites JavaScript invocations of WebView interfaces in the web page being rendered. It extends the parameter list of such a invocation to include a string that indicates the origin of the JavaScript (more details in § 4). The integrity and confidentiality is guaranteed by the enforcement of the same-origin policy inside WebView. Besides enabling origin-based policy enforcement, origin tagging also ensures that distinct WebViews within WIREFRAME cannot introspect on each other. For example, WIREFRAME intercepts WebViews' access to the local file system (via URI loading override) and transparently redirects such access to per-origin private paths, unless a client app defines a less restrictive policy.

Developers can take advantage of origin tagging to define custom policies, placed in the app's manifest. An app-defined policy object follows the same format as that of a web-defined policy object: (S_i, P_i). But in this case, the security principal S_i is a web origin and the policy P_i is a list of local interfaces that the app exposes to S_i. For example, a legitimate location service app can define a policy whose S_i is the app's own domain and P_i contains a local Java interface `getGpsLocation`, which returns the GPS location. This policy informs WIREFRAME that only web elements from origin S_i are allowed to invoke `getGpsLocation` via the app-web bridge whereas web elements from other origins, even if loaded inside the same WebView, are disallowed.

Such policies enable app developers to expose sensitive interfaces solely to intended web origins, which is a missing capability in today's WebView that causes the web-to-app attacks discussed in §2.2. With this capability, app developers no longer have to bear the high security risks while adding local support to their own or trusted web services.

3.2.3 Policy Sources

Policies can come from several different entities: a site can provide a policy when it is visited from an app, a developer can embed a policy into the app, and an expert user can even inject a policy into the app at rewriting time. For most apps and sites, the policy writer will have a notion of the type of behavior that they would like to enable or disallow. However, a potential exception arises when a developer uses a 3rd party SDK with web-embedding app functionality. In principle, the 3rd party should supply its own policy. Otherwise, developers can deploy a permissive policy that will still preserve SDK functionality.

4. WIREframe TECHNICAL DETAILS

In the previous section, we described the high-level protection mechanisms of our system. We now discuss the implementation of the runtime component, WIREframe, and show how it achieves the security goals introduced above. WIREframe is implemented as standalone third-party app that acts as a secure and trusted provider of WebView for regular apps. WIREframe completely mediates all interactions between an app and its embedded web content while enforcing fine-grained security policies.

Internally, WIREframe wraps one or more default WebView instances and use them to service an app's requests for WebView features. Apps make such requests and receive results via well-defined IPC interfaces exposed by WIREframe. Each IPC interface corresponds to a public WebView API and provides the equivalent functionality, except that it performs comprehensive security checks and enables policy enforcement. When in operation, WIREframe overlays its WebView UI on top of the invoking app's UI in the exact area where the original WebView is expected, providing a consistent and seamless user experience (i.e., the user is not aware of a web-embedded UI is in fact composed and supported by two separate apps). To keep the UIs of both apps synchronized, WIREframe and the client app collaborate to captures user interaction events (i.e., touches) and ensure that the proper UI receives the event based on its position.

We developed WIRE to automatically patch the proxy library into legacy apps and refactor the usage of WebView into IPC invocations to WIREframe without any developer assistance (WIRE is discussed in §A). Therefore, our system can be easily and quickly adopted in practice. An advantage of this deployment is that a developer or an end user can transition an app from using WebViews to using WIREframe mediation easily. It also allows for deploying regular WebViews and WIREframe side-by-side. We discuss the security implications of this deployment further in §6.

In the remainder of this section, we discuss the implementation of WIREframe by discussing how it handles the key challenges in its design.

Serialization: Android requires that objects passed via IPC have methods to handle their internal data marshalling and unmarshalling by implementing the `Parcelable` or `Serializable` interface. A few complex class types referenced in the WebView APIs do not implement these interfaces, and therefore, cannot be passed via IPC. Although data marshaling for IPC is a well studied problem, the unique constrains that we faced in designing WIREframe make the existing solutions non-applicable. For instance, adding serialization support to complex class types is not feasible without changing WebView or Android middleware. Furthermore, even if serialization methods can be added, a type may have volatile state that prevents if from being fully serialized or passed across app boundaries. In other words, such objects are inherently bound to their app contexts.

We handle unserializable types using a technique we call *object shadowing*. The intuition behind object shadowing is that, if an object cannot be moved to, or duplicated in, the remote process, we keep it in the original program context while creating a shadow copy of the object in the remote process. The shadow object acts as a transparent proxy to the original object: it only contains the public interfaces of the original object. The shadow copy's implementation of these interfaces simply invokes the corresponding interface exposed by the original object via IPC. As a result, the shadow object allows code in the remote process to invoke public methods or access public fields as if the original object were passed to the remote process. At the same time, when its methods are invoked the original object functions properly without suffering from broken dependencies that would otherwise occur if the object had been copied or duplicated in the remote process. Figure 4 shows an example of applying object shadowing to the second parameter of `WebView.evaluateJavascript`, a `ValueCallback` object. In the example, the original object, `callback` is kept at the client app side while a shadow object, `shadowCallback`, is automatically created in the WebView instance in WIREframe. The shadow object forwards calls to the public interface, `onReceiveValue`, back to the original object via the IPC channel provided by WIREframe.

Object shadowing can be recursive when a shadow interface takes or returns complex objects. The recursion is bounded due to the fact that object interfaces always converge to primitive types that can be directly transferred over IPC. The generation of these objects and classes is straightforward and automated. Thanks to object shadowing, non-serializable objects involved in WIREframe IPC interfaces are invoked in their original app context, rather than copied across app boundaries, which allows IPC-unfriendly objects to be used in a cross-app fashion.

Visual Fidelity: WebViews running in WIREframe need to appear and function as native UIs of their embedding apps. This includes not only displaying at the same scales and locations as native WebViews but also responding to events, for instance, indicating device rotation from landscape mode to portrait mode, in which case the content rendered in the WebView should automatically rotate and resize. Simply using the floating UI feature of Android does not enable synchronization among the UIs belonging to two apps. For instance, when the device is rotated, a series of events is sent down the view hierarchy of the embedding app, updating the layout of each element. This context is not available to the WIREframe and is necessary to calculate the final position and size that the WebView would have occupied.

```
/* An example of a WebView API involving
complex parameter */
void evaluateJavascript(
     String script,
     ValueCallback<String> resultCallback)
```

```
/* Pseudo-def of a ValueCallback object in the
original app context */
ValueCallback<String> callback =
new ValueCallback<String>(){
     @Override
     public void onReceiveValue(String value) {
          ... // Original callback handler
     }};
```

```
/* Pseudo-def of a shadow ValueCallback object
in the WIREframe app */
ValueCallback<String> shadowCallback =
new ValueCallback<String>(){
     @Override
     public void onReceiveValue(String value) {
          // Auto-generated IPC stub
          WIREframeIPCAgent.remoteInvoke(
                    OrigValueCallBackObjID,
                    value);
     }};
```

Figure 4: An illustration of object shadowing

To achieve visual fidelity, the Proxy WebView maintains an invisible view (i.e., a transparent placeholder) that takes the size and shape of the original WebView and forwards all view events to the WIREframe via IPC. Android supports several types of floating UI, by which an app in the background can draw UI elements on top of the currently foregrounded app. We leverage the floating UI feature to place the trusted WebView managed by WIREframe over the rewritten app while the latter is running in the foreground. The WIREframe WebView occupies the exact screen area where the original WebView would have been rendered had the app not been rewritten or WIREframe not deployed.

Origin-based Policy Enforcement: To achieve fine granularity, our policy enforcement needs to track the origins of web content and the origins of web-initiated calls to the app-web bridge. Without this capability, WIREframe cannot enforce useful policies such as allowing only a particular origin to invoke the GPS-reading method exposed by a client app. However, realizing this capability in WIREframe is challenging because none of the WebView APIs are aware of the notion of web origins (i.e., their parameters and return values do not carry information about origins).

In order to retain the origin information for each web-to-app data access or code invocation, WIREframe employs a dynamic HTML rewriting technique, which we call *origin tagging*. This technique is built on the standard WebView callbacks that the embedding app (WIREframe in this case) can register to handle web navigation events. Upon each page (re)load or DOM element refresh event, WIREframe receives a callback from WebView's rendering event inspector. During this callback, WIREframe rewrites every Javascript-to-WebView invocation in the to-be-loaded page by appending an origin label to the parameter list. WIREframe then resumes the page process. Using the origin tagging technique, WIREframe attaches the genuine origin labels to the invocations of the app-web bridge in a webpage before the page is loaded. Any obscured invocation that is not labeled will be rejected by the Policy Checking during invocation. Note that an origin label is an encoded string that can only be decoded into a plain origin string with the secret key randomly generated key for the current webpage. The encrypted labels prevent malicious web content from faking or tampering with their origin labels. Later on when a rewritten invocation is triggered, the Policy Checker retrieves the origin label by inspecting the last parameter of the call. It decodes the label, verifies its integrity, and then checks the invocation against the origin-based policy.

Complete Mediation: An important guarantee that our system provides is that *all* app-web interactions are subject to policy enforcement. However, there is an inherent difficulty in maintaining this guarantee without modifying the Android framework: An adaptive adversary may attempt to hide the use of a default WebView from rewriting by WIRE, or may re-implement web-embedding features in third-party code. To address these potential evasions, WIREframe allows sensitive websites to require that they must be accessed from WIREframe. A website expresses this requirement when WIREframe requests a specific path of a website (e.g., example.com/wireframe.txt). WIREframe makes such a request and checks a website's requirement when the website is to be contacted directly by a client app. The result is cached to save repetitive checks. We imbue WIREframe with the ability to intercept all network traffic from and to client apps. When WIREframe observes a client app directly requesting HTTP or HTTPS content from a website that requires WIREframe, it blocks the request and alerts.

We realize this feature using the `VpnService` class, which allows an app to act as a VPN client without requiring root privilege. While the intended usage of the class is for building a tunnel interface, we repurpose it for traffic interception on client apps of WIREframe. By implementing a *per-app VPN*, WIREframe can force the client apps to send all traffic through it while not affecting other app's network connections. The per-app VPN can co-exist with other VPN client apps.

Note that using a mediated tunnel in this way leverages a key advantage of our approach: WIREframe and its client-side library set up the secure service at the entry points and torn down at the exits of a client app. The complete mediation enforced by the secure service ensures that any WebViews missed by WIRE are detected at runtime.

5. APP REWRITING

A key goal of our work is that it is backwards-compatible with existing apps without modifying the OS. In this section, we discuss the details of how we accomplish this goal through app rewriting.

The security policies discussed in Section 4 only take effect if WIREframe is used by a web-embedding app in place of its regular WebViews. While benign developers might choose to deploy our mechanisms, malicious developers have no incentive to do so. Our offline rewriting tool, WIRE, addresses this concern by replacing all uses of WebView with uses of the secure WIREframe proxy. This section provides details on the design and implementation of WIRE. As space constraints prevent us from providing a full discussion of the implementation of WIRE, we highlight the novel and challenging aspects of the tool.

Packaged App Analysis: One of the key advantages of our approach is that it does not require assistance from developers. This means that the tool can rely only on the packaged app (.apk file) and compiled bytecode. To handle this challenge, WIRE leverages previous work on reverse-engineering and re-compiling Dalvik bytecode. In particular,

we use the open source Apktool to unpackage and repackage code and resources from an apk [1]. We use the Soot Java Optimization Framework [30] and Dexpler [4] to extract Dalvik to an intermediate representation and recompile the rewritten code.

WIRE is designed as a modular pipeline, with the rewriting phase decoupled from unpackaging and repackaging the app. Thus, improvements to the underlying tools can be easily integrated into our workflow.

Identifying WebView Usage: Because WIREFRAME prevents the use of the default WebView, it is crucial for the proper operation of the client app that all legitimate WebView uses of are identified and replaced. Unfortunately, this identification can be challenging. In addition to WebViews that are programmatically constructed and configured at runtime, an app can define the WebView UI and its layout using an XML manifest which the system loads at runtime. Thus, WIRE introspects and modifies not just the app code, but also the applications resource XML files and support code. We provide additional details of our rewriting algorithm, including pseudocode, in Appendix A.

Satisfying Lifecycle Constraints: Android apps execute in an event-driven *lifecycle* managed by the system. Events are fired by the Operating System in response to events or system notifications. An implicit ordering exists between the lifecycle events: one event cannot happen until the component's lifecycle has gone through preceding events. Without considering component lifecycle and the implicit constraints, app rewriting can cause erroneous or interrupted app execution. Thus, WIRE includes a model of the Android lifecycle, which is referenced during the calls to inject and marshall so that the WIREFRAME is properly running and bound before each invocation.

6. SECURITY ANALYSIS

We now discuss the security and robustness of our system against evasion. Our discussion concerns attacks launched by either a malicious client app or a malicious webpage—two types of adversaries allowed in our threat model. We explain how our design addresses each adversary, and discuss limitations of our approach.

Malicious client apps: Adversarial apps may attempt to evade our bytecode rewriting process to maintain the usage of an unprotected WebView, and in turn preserve an attack on the WebView's content. A sufficiently advanced adversary may be able to evade WIRE through obfuscation (e.g., using Java reflections), native code, or dynamically loaded code. However, the per-app `VPNService` implemented in WIREFRAME blocks any traffic, including HTTPS, from a client app (i.e., using obfuscated WebView) to a server that requires the use of WIREFRAME (this requirement is declared in the server's response headers and can be independently tested by WIREFRAME). This behavior highlights the fail-safe nature of our system: if a hidden web connection avoids WIRE, it will cause the app to break rather than obtaining unmediated web access.

Malicious apps may hijack the IPC channel through which the client-side proxy and WIREFRAME communicate, leading to unchecked or forged WebView API calls. The adversary may employ IPC spoofing (i.e., communicating to WIREFRAME directly without going through the local proxy) or compromise the local proxy. The client app is considered as a single, untrusted entity from the perspective of the web content, and all calls to the IPC interface are mediated on the WIREFRAME side. In other words, the WIREFRAME treats all apps as if they were under the control of an adversary spoofing the local proxy.

App Name	Category	Functional	Visual
Dictionary.com	Reference	✓	✗
Flappy Bird	Entertainment	✓	✓
Facebook	Social	✓	✓
LinkedIn	Social	✓	✓
The Hindu	News	✓	✓
NY Times	News	✓	✓
The Economic Times	News	✓	✓
Groupon	Social	✓	✓
IMDB	Reference	✓	✓
Amazon Shopping	Shopping	✓	✓
Ebay	Shopping	✓	✓
Textgram	Social	✓	✓
Jewels Saga	Entertanment	✓	✓
Ask.fm	Social	✓	✗
Photodirector	Media	✓	✓
Angry Birds	Entertainment	✓	✓
Instant Inventory	Shopping	✓	✓
Fun Run	Entertainment	✓	✓
LivingSocial	Social	✓	✓
QuickPic	Media	✓	✓

Figure 5: Table of selected benign apps rewritten using WIRE. A ✓ indicates that the given app uses an overlay over a WebView, while a ✗ indicates that the given app does not.

Malicious web content: When rendered inside WIREFRAME, a malicious web page may attempt to break the isolation and security checks enforced by the trusted WebView. Since the web origin plays a central role in regulating untrusted web content, the origin tagging mechanism of WIREFRAME can be an obvious target for attackers. For example, malicious JavaScript can either obfuscate its invocation of Java interfaces to avoid tagging, or spoof its origin by stealing a tag assigned to scripts from other domains. Although it is possible to hide Java invocations, such invocations are rejected by WIREFRAME as they are not tagged. Stealing tags is impossible because reading tags of scripts from other domains is prevented by the SOP. Moreover, origin tags cannot be forged or reused because they are randomly generated on a per session basis. In very rare cases, attackers may successfully exploit vulnerabilities in the web rendering engine, and possibly compromise the TCB of WIREFRAME. While not designed to mitigate such low-level attacks, our system does significantly reduce the potential damage that such attacks can cause to either client apps or WIREFRAME thanks to the process-based separation of each WebView instance.

7. EVALUATION

Our evaluation seeks to answer the following questions:

1. *Correctness*: Do apps have the same appearances and functionalities after adopting WIREFRAME?
2. *Effectiveness*: Does WIREFRAME enforcement effectively prevent attacks on the app-web bridge?
3. *Efficiency*: What is the performance impact of replacing in-app WebViews with WIREFRAME?

Experimental Highlights: Our experiments validate our approach and show encouraging results. All apps, of different categories, continued to run correctly after being rewritten using WIRE to use WIREFRAME, with 90% showing

no visual differences at all. We found that WIREFRAME effectively prevents both web-to-app and app-to-web attacks: WIREFRAME successfully stopped the attacks against four popular third-party WebView libraries that were otherwise vulnerable, and prevented real web exploits targeting apps found in the wild.

In the remainder of this section, we describe our methodology for arriving at these conclusions, and provide a more in-depth analysis of our results.

7.1 Methodology

To answer the evaluation questions posed above, we use both apps found in the wild and synthetic examples specifically crafted to highlight particular aspects of our approach. As has been stated in previous work, scaling an evaluation to a large number of apps found in the wild is difficult because apps are highly interactive [8]. Thus, we take a similar approach to contemporary work: we statically assess a large corpus of 7166 apps, then select a subsample of 20 relevant, representative apps for deeper manual inspection. We refer to the statically tested set as the *correctness apps*, since the static analysis focuses on the validity and correctness of the rewriting. We refer to the 20-app subsample as the *benign apps* since we exercise the benign behavior of the apps with the goal of ensuring that the app still functions correctly. We note that the size of these samples are similar, for example, to [32], which used a static sample of 1612 apps and a targeted subsample of 20. We note several special-purpose analyses test against even smaller sets, such as [22], which manually analyzes 7 apps.

In addition to our samples of apps found in the wild, we also create two sample sets designed to test the security and performance of our approach: a set of *attack apps* that we designed to mount attacks against 4 popular third-party WebView libraries, and a set of *benchmark apps* for precisely measuring the performance of our approach. Designing synthetic tests allows for repeatability, since the apps can be built to run with minimal interaction with the user and in a deterministic way.

We now describe each of our sample sets in greater detail:

Correctness Apps: To ensure the external validity of WIRE, we applied it to a collection of 7166 app downloaded from Google Play and 3rd party markets. Given the size of this sample, running each app manually is infeasible. Our goal with this sample is to ensure that the transformations applied by WIRE are correct and produce valid bytecode even on apps found in the wild.

Benign Apps: Our suite of benign apps is composed of 20 popular apps specially selected from the Google Play store, sub-sampled from the correctness apps. The apps come from a variety of categories including reference (for reference material, such as a dictionary), entertainment (for games), Social (for social content such as Facebook), and Media (for traditional media apps such as image viewers). Figure 5 shows the selected benign apps, along with their categories.

Attack Apps: Our set of attack apps exploits WebViews used in four popular third-party libraries: LinkedIn, Facebook, Twitter, and Foursquare. The basic flow of the attack is very similar to that of our example attack, WEBRSS, discussed in Section 3. Each attack app creates a WebView and uses the API of the third-party library to get a sign-on URL from the associated provider. The attack app then injects JavaScript into the login page to read the username and password fields on that page.

To apply the extra security protection of a login page, WIREFRAME needs to know when it is on a secure login site. In a production system, the secure web page would provide a dynamic policy to indicate to WIREFRAME that the page should selectively allow JavaScript to be injected or any web content to be introspected upon. However, in our experiment, instead of altering the HTML headers of the login page and install dynamic policy on behalf of the SSO providers, we simply rely on the default and the most restrictive policy of WIREFRAME: by default, without cooperation from the site, WIREFRAME does not allow apps to inject scripts to or inspect on embedding WebView.

Benchmark Apps: To characterize per-operation overheads associated with WIREFRAME, we manually insert timing checks into a set of synthetic apps. We are broadly interested in three measures of overhead: the space cost of having an additional app on the device, the per-launch overhead of establishing the communication channel between client apps and the WIREFRAME services, and the per-use overhead of the IPC-based interaction between a client app and its embedded WebView.

7.2 Analysis

7.2.1 Correctness

We performed two experiments to ensure the correctness of our approach. In the first, we ensured that the app rewriting performed by WIRE produced valid bytecode. In total, we found that 46 of our 7166 apps (approximately 0.6% of apps) failed to complete the rewriting successfully. We note that all of these apps also fail to complete a null transformation in Soot (our underlying analysis engine). Thus, we believe these limitations are not intrinsic to our technique.

In our second correctness experiment, we tested that the apps in our benign sample of apps continued to perform correctly when run manually. Figure 5 shows the results of this experiment on our 20 web-embedding apps.

Functional Correctness: The *Functional* column indicates that the functionality of the app was preserved: no crashes were detected in a manual session of operating the app, and all web and app tasks completed using the WIREFRAME just as using a plain WebView.

Visual Fidelity: The *Visual* column of Figure 5 indicates if the app using WIREFRAME versus the in-app WebView appeared to be identical. We discovered none but two apps that did not meet this criteria, which were expected corner cases. As a security feature, WIREFRAME does not allow client apps to overlay UI over any part of WebView, and therefore, prevents clickjacking and other UI confusion attacks. The 2 apps failed the visual fidelity test because of this deliberate security restriction of WIREFRAME. In the `Ask.fm` app, a loading widget from the app is placed over the WebView while it loads, and is thus not visible in the rewritten app. In the `Dictionary.com` app, a widget from the app displays an advertising message for a premium version of the app over web content. In both cases, the workflow of the apps remain undistorted. Furthermore, these offending overlays could have been embedded directly into the web content or displayed elsewhere in the apps.

Tested API		API Invocation Time (in milliseconds)				API Invocation Overhead (relative)	
		w/ WIREframe		w/o WIREframe			
Name	*Type*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*
`clearCache`	basic	2.38	2.23	1.22	0.82	**0.95**	**1.72**
`getTitle`	basic	0.58	0.183	0.30	0.11	**0.93**	**0.72**
`capturePicture`	complex	6.08	6.97	1.16	1.64	**4.25**	**3.24**

Figure 6: Runtime Overhead of WIREframe protection mechanisms. Overhead includes the IPC invocation and policy checks. Note that the complex object shadowing of `capturePicture` includes the time needed to copy an entire screenshot of a WebView between apps.

	Time				Relative Overhead	
	w/ WIREframe		w/o WIREframe			
	Nexus 5	*Samsung S5*	*Nexus 5*	*Samsung S5*	*Nexus 5*	*Samsung S5*
Load URL w/ origin tagging (ms)	13.24	15.16	12.63	14.30	**0.05**	**0.06**
Load URL w/o origin tagging (ms)	12.38	14.43	12.63	14.30	**-0.02**	**0.01**
Average app boot and load (s)	5.37	6.12	5.09	4.68	**0.05**	**0.08**

Figure 7: Runtime Overhead of the Load URL API. Note that loading URLs without origin tagging has a low enough overhead that it is within the margin or error.

	w/ WIREframe		w/o WIREframe	
	N5	*S5*	*N5*	*S5*
Client – Kernel time (s)	0.6	0.3	0.8	7.6
Client – User time (s)	1.8	1.1	8.7	3.7
Wf – Kernel time (s)	0.7	2.4	-	-
Wf – User time (s)	3.7	9.6	-	-
Client – VSS (KB)	945	965	1021	1061
Client – RSS (KB)	66.6	37.4	72.7	100
Wf – VSS (KB)	947	952	-	-
Wf – RSS (KB)	46.7	47.1	-	-

Wf = WIREframe App

Figure 8: Resource Utilization of CPU and Memory. WIREframe incurs modest overhead, mostly composed of time and memory in user space.

7.2.2 *Effectiveness*

For each of the four attack apps that we tested, we found that WIREframe was effective in preventing the malicious behavior that we inserted.

Effective Enforcement: The attack apps import and exploit the authentication libraries from Facebook, Foursquare, LinkedIn, and Twitter, all of which use WebViews. To exploit the library, the attack apps inject JavaScript into the login window for each service according to the techniques described in Section 3. For all four libraries, we successfully extracted the username and password when the app used a default in-app WebView. We then rewrote each app using WIRE, and replayed the attacks. In each case, WIREframe successfully prevented exfiltration of credentials.

In addition, we simulated the web-to-app attacks and examined WIREframe's origin-based policy enforcement. We created a test app which, employing dynamic policies, exports a range of sensitive Java interfaces exclusively to web content from a trusted origin. We also composed a mash-up page with multiple iframes and scripts from different origins that all try to access the exported app-web interfaces. During the test, the app first loads the mash-up page using a regular in-app WebView and then do the same using WIREframe. Our results show that, the sensitive interfaces were universally accessible to all web content loaded in the regular WebView but were only accessible to the trusted domain from within WIREframe.

Those tests show that WIREframe's enforcement is effective at isolating the threats that apps and embedded web content may impose to each other.

7.2.3 *Efficiency*

The extra security protections afforded by our approach have overheads in terms of resource utilization (CPU and memory) and runtime overhead. While correctness and effectiveness are the primary concerns of our system, we also evaluate if the mechanism is efficient enough to use.

Resource Utilization: Figure 8 lists the resources used by an app with and without WIREframe. VSS lists the virtual set size (VSS), which is a measure of the maximum utilization of virtual memory. RSS lists the resident set size, which measures the maximum footprint in resident memory. App using WIREframe has a smaller memory footprint across both metrics, since web content is now being loaded in the WIREframe process. There is also a constant overhead of less than 1 MB for running the additional process, but given that modern Android devices such as the S5 are equipped with 2GB of RAM, we consider this overhead to be negligible.

Runtime Overhead: Rewritten apps incur overhead from the extra bookkeeping performed for WIREframe protection mechanisms. We measured the runtime increase across representative web APIs of both types.

Figure 6 shows the runtime of invocations of two *basic* APIs, in which the arguments to the call do not require object shadowing and *complex* APIs which do. These functions measure the additional overhead of app to-web protections, which is accounted for by the actual IPC invocation and related marshaling. For basic APIs, we experience an approximately 1x increase in overhead. For complex APIs, we experience a 3-4x increase.

Figure 7 shows the overhead of loads with and without origin tagging. This overhead is accounted for by building and inspecting the web origin. As expected, we experience negligible overhead without origin tagging (within the margin of error of our timing tool, `DDMS`).

Although these overheads are high in relative terms, they are mitigated by the fact that the absolute overheads are small. Given that these WebView APIs are called infrequently in an app, the runtime overhead accounts for a negligible factor of the total runtime of the app. We have found these latencies to be acceptable in use, but we note that there is room to optimize our techniques, especially with regards to object shadowing. Furthermore, interacting with web content is especially amenable to absorbing the overheads introduced by IPC, since runtime of such operations will often by dominated by network latency.

8. RELATED WORK

Studying WebView-related Attacks: Previous studies have reported several types of WebView attacks that exploit the app-web bridge. Luo *et al* [16] demonstrated that, using WebView APIs, apps may inject malicious scripts into embedded web content, and at the same time, unauthorized web code may invoke app-exported Java methods. Roesner *et al* have noted that apps can read passwords from the embedded WebViews [24]. Many works have noted the scope and severity of malicious web content on benign apps (*web-to-app* attacks, in our terminology: Chin *et al* [6] studied two types of WebView attacks whereby malicious JavaScript scripts perform unauthorized Java invocations and file system access in vulnerable apps. Neugschwandtner *et al* [19] showed that WebViews can serve as a powerful attack vector when the server is compromised. Thomas *et al* [28] formulated a model for determining the lifetime of a vulnerabilities in Android using Javascript attacks on WebView as a case study. This model notes the slow deployment of patches in Android, a point that supports our technique of app rewriting rather than system WebView patching. Wang *et al* [31] demonstrated the origin-confusion attacks and provided a mitigation that requires OS modifications. More recently, Son *et al* [27] found that untrusted advertisements rendered in WebViews may infer user profiles by testing the mere existence of certain files, an operation that the current WebView design cannot forbid. Motivated by those previous studies, our work solves an open and pressing issue—generalizing and preventing WebView-related attacks.

Isolating External Web in Apps: There is a rich body of work [10, 20, 26, 33] on mobile ads isolation. The proposed solutions isolate ads from hosting app by placing ads in a separate process or app. NativeWrap [18] expands the similar isolation to cover web applications in WebViews. Our work also uses process boundaries to separate apps and web content, but is compatible with all kinds of WebView usages and considers both web-to-app and app-to-web attacks. Unlike previous work, our system allows for policy-driven and origin-based security, and includes a static rewriting tool, WIRE, to help app users conveniently apply WIREframe to existing apps that use WebView. Draco [29] is the latest work that mitigates untrusted web content rendered in WebView by extending the WebView system app on recent versions of Android. In comparison, our work does not require rooted devices or deployment assistance from OS or device vendors. Our work applies to both web-to-app attacks and app-to-web attacks, which previous work cannot.

Securing Sensitive Web Content in Untrusted Apps: Web-based logins are a common embedded web element that previous research set to secure [5, 15, 25] by means of trusted devices, verified UI, and scrutinized implementation of authentication protocols. In contrast, WIREframe prevents the web content manipulations unique to WebView. Such manipulations are caused by the faulty security assumptions of WebView and the coarse security control over the app-web bridge. LayerCake [24] is a modified version of Android that prevents UI confusion and clickjacking attacks. It supplies secure user interfaces elements, including SecureWebView that can be embedded in an app but run in a separate process. SecureWebView statically disallows the use of JavaScript and the app-web bridge. Therefore, it can prevent the SSO attacks that partly motivated our work. However, SecureWebView only aims to protect sensitive web content whereas WIREframe protects both apps and web content as per the policies from both sides. In addition, WIREframe is backward compatible with the existing Android architecture While the goals of our systems are different, it would be interesting to combine the systems: LayerCake could enable the app-web bridge but enforce the policies that we describe in this paper, and WIRE could retarget legacy apps to use the OS-provided SecureWebView. An alternative approach used by Mutchler *et al* [17] and Hassanshahi *et al* [11] is to scan web-embedding applications offline for possible web-app bridge vulnerabilities. While these papers do not specifically mention SSO credential stealing, they share a similar threat model to our own in that they consider malicious apps as well as malicious web traffic. Unlike our work, these techniques do not propose defenses other than reporting the possible vulnerabilities.

Hybrid Frameworks: Frameworks such as PhoneGap / Cordova [3] allow developers to write apps in web languages, including HTML and JavaScript. The abstractions provided by such frameworks could implement some of the protections against malicious web content that we describe. For example, Cordova can hook URL loading and inject filtering. However, it is the responsibility of the developer to use the framework correctly, and thus enforcement is not mandatory. Some recent works [13, 9] attacked hybrid apps via local code injection or remote resource abuse. They proposed mitigations that are specific to hybrid apps and require changes to the frameworks. In comparison, WIREframe focuses on native apps that embed web content. Since the hybrid frameworks all use WebView as their building blocks, they may in principle adopt WIREframe's policy-driven, origin-based security model to govern web elements in hybrid apps.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This project was supported by the Army Research Office (Grant#: W911NF-17-1-0039), the National Science Foundation (Grant#: CNS-1421824, CNS-1228782, and CNS-1228620), and a joint United States Air Force/-DARPA Contract (# FA-8650-15-C-7562). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

10. CONCLUSION

As discussed in this work and others, Web-embedding apps increasingly attract attacks from different angles. Several current threat vectors remain unprotected, due to the lack of practical security mechanisms that can meet security requirements of all parties, including app developers, app users and web content providers.

We propose the use of a secure, third-party app called WIREframe to provide trustworthy web-embedding while enforcing configurable and origin-based security policies on the interactions between Android apps and embedded web content. WIREframe allows both apps and web content to secure their own resources at fine-granularities. We have shown that our solution is effective in preventing abuses of the app-web bridge by either malicious web content or malicious apps. At the same time, our system maintains the appearance and functionality of client apps.

Our solution is easy to deploy. It requires no modification to the Android operating system or framework. Through the use of our offline app-rewriting tool, WIRE, we can retarget legacy apps to benefit from the enhanced security of WIREframe without developer intervention.

11. REFERENCES

[1] *Android-Apktool*. https://ibotpeaches.github.io/Apktool/.

[2] *Android Isolated Service*. http://developer.android.com/guide/topics/manifest/service-element.html#isolated.

[3] *Apache Cordova*. https://cordova.apache.org.

[4] A. Bartel, J. Klein, *et al. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot*. Proceedings of the 1st International Workshop on the State Of the Art in Program Analysis, SOAP '12. ACM, 2012.

[5] E. Bursztein, C. Soman, *et al. Sessionjuggler: Secure Web Login from an Untrusted Terminal Using Session Hijacking*. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '15, 321–330. ACM, 2012.

[6] E. Chin & D. Wagner. *Bifocals: Analyzing WebView Vulnerabilities in Android Applications*. In *Information Security Applications*, LNCS, 138–159. Springer International, 2014.

[7] J. Dean, D. Grove, *et al. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*. ECOOP '95, Berlin, Heidelberg.

[8] M. Egele, C. Kruegel, *et al. PiOS: Detecting Privacy Leaks in iOS Applications*. In *Proceedings of the 2011 Network and Distributed System Security Symposium*, NDSS '11, 177–183. 2011.

[9] M. Georgiev, S. Jana, *et al. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks*. 2014.

[10] M. C. Grace, W. Zhou, *et al. Unsafe Exposure Analysis of Mobile In-app Advertisements*. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC 12, 101–112. ACM, 2012.

[11] B. Hassanshahi, Y. Jia, *et al. Web-to-Application Injection Attacks on Android: Characterization and Detection*. In *Proceedings of the 2015 European Symposium on Research in Computer Security*, ESORICS '15, 577–598. Springer, 2015.

[12] J. Jeon, K. K. Micinski, *et al. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications*. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 3–14. 2012.

[13] X. Jin, X. Hu, *et al. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation*. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, CCS '14, 66–77. ACM.

[14] G. A. Kildall. *A Unified Approach to Global Program Optimization*. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, 194–206. ACM, 1973.

[15] D. Liu & L. P. Cox. *VeriUI: Attested Login for Mobile Devices*. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 7. ACM, 2014.

[16] T. Luo, H. Hao, *et al. Attacks on WebView in the Android system*. In *Proceedings of the 2011 Annual Computer Security Applications Conference*, 343–352. ACM, 2011.

[17] P. Mutchler, A. Doupé, *et al. A Large-Scale Study of Mobile Web App Security*. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*. 2015.

[18] A. Nadkarni, V. Tendulkar, *et al. NativeWrap: Ad Hoc Smartphone Application Creation for End Users*. In *SPWM 2014*, WiSec '14, 13–24. ACM, 2014.

[19] M. Neugschwandtner, M. Lindorfer, *et al. A View to a Kill: WebView Exploitation*. In *LEET 2013*. USENIX, 2013.

[20] P. Pearce, A. P. Felt, *et al. Addroid: Privilege Separation for Applications and Advertisers in Android*. In *SICCS 2012*.

[21] V. Rastogi, R. Shao, *et al. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces*. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16. 2016.

[22] B. Reaves, N. Scaife, *et al. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World*. In *Proceedings of the 24th USENIX Security Symposium (2015)*, 17–32. 2015.

[23] T. Reps, S. Horwitz, *et al. Precise Interprocedural Dataflow Analysis via Graph Reachability*. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT POPL Symposium*, POPL '95, 49–61. ACM, 1995.

[24] F. Roesner & T. Kohno. *Securing Embedded User Interfaces: Android and Beyond*. In *Proceedings of the 22nd USENIX Security Symposium*, Security '13, 97–112. USENIX, 2013.

[25] M. Shehab & F. Mohsen. *Towards enhancing the security of oauth implementations in smart phones*. In *ICMS 2014*, 39–46. IEEE, 2014.

[26] S. Shekhar, M. Dietz, *et al. AdSplit: Separating Smartphone Advertising from Applications*. In *USENIX Security Symposium*, 553–567. 2012.

[27] S. Son, D. Kim, *et al. What Mobile Ads Know About Mobile Users*. In *NDSS*. 2016.

[28] D. R. Thomas, A. R. Beresford, *et al. Security Protocols XXIII: 23rd International Workshop, Cambridge, 2015*, 126–138. Springer International, 2015.

[29] G. S. Tuncay, S. Demetriou, *et al. Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android*. In *Proceedings of the 2016 Conference on Computer and Communications Security*, CCS '16, 104–115. ACM, New York, NY, USA, 2016.

[30] R. Vallée-Rai, P. Co, *et al. Soot - a Java Bytecode Optimization Framework*. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99. IBM Press, 1999.

[31] R. Wang, L. Xing, *et al. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation*. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 635–646. ACM, 2013.

[32] L. Xing, X. Bai, *et al. Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS*. In *Proceedings of the 2016 Conference on Computer and Communications Security*, 31–43. ACM, 2015.

[33] X. Zhang, A. Ahlawat, *et al. AFrame: Isolating Advertisements from Mobile Applications in Android*. In *Proceedings of the 2013 Annual Computer Security Applications Conference*, ACSAC '13, 9–18. ACM, 2013.

APPENDIX

A. APP REWRITING

A key goal of our work is that it is backwards-compatible with existing apps without modifying the OS. In this section, we discuss the details of how we accomplish this goal through app rewriting.

The security policies discussed in Section 4 only take effect if WIREframe is used by a web-embedding app in place of its regular WebViews. While benign developers might choose to deploy our mechanisms, malicious developers have no incentive to do so. Our offline rewriting tool, WIRE, addresses this concern by replacing all uses of WebView with uses of the secure WIREframe proxy. This section provides details on the design and implementation of WIRE. As space constraints prevent us from providing a full discussion of the implementation of WIRE, we highlight the novel and challenging aspects of the tool.

Packaged App Analysis: One of the key advantages of our approach is that it does not require assistance from developers. This means that the tool can rely only on the packaged app (.apk file) and compiled bytecode. To handle this challenge, WIRE leverages previous work on reverse-engineering and re-compiling Dalvik bytecode. In particular, we use the open source Apktool to unpackage and repackage code and resources from an apk [1]. We use the Soot Java Optimization Framework [30] and Dexpler [4] to extract Dalvik to an intermediate representation and recompile the rewritten code.

WIRE is designed as a modular pipeline, with the rewriting phase decoupled from unpackaging and repackaging the app. Thus, improvements to the underlying tools can be easily integrated into our workflow.

Identifying WebView Usage: Because WIREframe prevents the use of the default WebView, it is crucial for the proper operation of the client app that all legitimate WebView uses of are identified and replaced. Unfortunately, this identification can be challenging. In addition to WebViews that are programmatically constructed and configured at runtime, an app can define the WebView UI and its layout using an XML manifest which the system loads at runtime. Thus, WIRE introspects and modifies not just the app code, but also the applications resource XML files and support code.

Figure 9 presents simplified pseudocode for the rewriting algorithm. The goal the main loop is to transform a target web-embedding app $\mathcal{A}$ to use the secure webview proxy instead of the default `WebView` class of Android. The first step of our algorithm (Line 2) is to invoke a custom extension to the standard class hierarchy analysis (CHA [7]), which we refer to as CHA'. This extension ensures that class loading from the resources (*i.e.* the app manifest) is also included. Once we have identified all instances and subclasses of `WebView`, denoted W, we create a proxy w' for each $w \in W$ (Line 4). The main rewriting loop (Lines 3-13) is concerned with substituting w with w': we identify the set the set U_w of all uses of $w \in W$ (Line 5) using a standard dataflow [14] over the supergraph of $\mathcal{A}$ [23]. Note that we use the term *use* to refer to all operations that reference w, as opposed to the typical use/def relations in which usually only refer to the right-hand side of an operation. For notational convenience, we represent each such use in the form $c(w, v_1, v_2, \ldots, v_n)$ where $v_1, \ldots, v_n$ are additional

```
 1: Let 𝒜 be the target web-embedding app
 2: W ← CHA'(WebView, 𝒜)
 3: for all w ∈ W do
 4:     Create proxy w' of w
 5:     Let U_w be the set of uses of w in 𝒜
 6:     for all c(w, v_1, ..., v_n) ∈ U_w do
 7:         replace(w, w')
 8:         for all i ∈ {1, ..., n} do
 9:             Let s_{u_w} be the shadow object of v_i
10:             S ← S ∪ s_{u_w}
11:     for all s_{u_w} ∈ S do
12:         marshal(s_{u_w})
```

Figure 9: Rewriting loop simplified pseudocode

variables involved in the use, such as arguments to methods of w. These additional variables are exactly the ones that need to be shadowed or serialized (c.f. section 4). Thus, we replace the reference to w itself with a reference to w' using the pseudofunction replace (Line 7) and keep an object s_{u_w} to shadow each v_i (Lines 9-10). Finally, all the Webviews have been replaced, we call the marshal pseudofunction with each shadow object to inject the marshaling code necessary to transfer the used value shadowed by s_{u_w} into the proxy (Lines 11-12).

Satisfying Lifecycle Constraints: Android apps execute in an event-driven *lifecycle* managed by the system. Events are fired by the Operating System in response to events or system notifications. An implicit ordering exists between the lifecycle events: one event cannot happen until the component's lifecycle has gone through preceding events. Without considering component lifecycle and the implicit constraints, app rewriting can cause erroneous or interrupted app execution. Thus, WIRE includes a model of the Android lifecycle, which is referenced during the calls to inject and marshal so that the WIREframe is properly running and bound before each invocation.

B. FUTURE WORK

In this section, we discuss limitations of our current implementation and consider future work to them.

App Updates: A consequence of using offline rewriting to induce enforcement mechanisms on apps is that apps can no longer be automatically updated on the device. This is an inconvenience for users who enable automatic updates, since they have to re-apply the WIRE rewriting. However, this inconvenience can be justified by the much enhanced security of web-embedding apps without requiring OS changes. Furthermore, we expect that WIRE will mostly be applied to legacy apps (which are updated less frequently) and untrusted apps that benefit from additional static checking before install time in any case. Apps that do not include WebViews or adopt WIREframe during development do not need to be rewritten. In cases where app markets can adopt WIRE and perform app rewriting before app release, such as in an enterprise app store, app users can enjoy the security benefits of WIREframe without facing app update inconvenience.

WebView State Sharing: As shown by the attack in §2, allowing multiple WebViews to run in the same process enables implicit sharing of states, such as history and cookies. WIREframe runs each mediated WebView in a separate process to disable cross-WebView attacks. It also restricts

each WebView's file system access to a per-origin private path by default. However, sharing states among WebView instances created by a same app may be required for legitimate functionalities. While we did not encounter any such cases in our experiments, WIREframe could be extended to allow multiple WebViews to share a process. We leave this implementation detail, and the design of when to allow sharing, to future work.

OS-level Extension of WebView: While one of the key contributions of our work is that it provides support for apps without updating the OS, this approach comes with a number of tradeoffs: the WIREframe service is isolated, but comes with the overhead of running a background service fulltime, as well as incurring the cost of object shadowing to communicate with the the client app. A natural alternative to our approach is to modify the OS directly to implement our proposed protections.

An obvious way to protect web-embedding app attacks is to extend the `WebView` class in Android framework to support a "trusted mode". Either an embedding app or embedded web content may switch a WebView instance into the trusted mode by calling newly introduced Java or JavaScript APIs. When in this mode, the web-embedding app runs the WebView, but the OS can suppress the app's introspection capabilities and dynamically regulating method invocations from embedded web content. In the Android security model processes are the atomic security principal [12]. Therefore, this approach is likely to require major changes to the security model of the OS, or at least rendering the trusted WebView in a separate process. Nevertheless, there are several benefits to this approach:

- **Performance improvements:** By implementing WebView isolation within the app, fewer context switches and less data marshalling is required.
- **Mandatory enforcement:** Our system is only effective if users apply the app rewriting tool WIRE or developers explicitly target WIREframe. By integrating WIREframe-like protections into the OS itself, web developers can be more confident that WebView policies are actually enforced on the client side.
- **Enhanced functionality:** As noted in Section 7, some visual differences may occur if the app attempts to "pop under" content on the WebView. While disallowing this behavior can help defend against clickjacking, it prevents a benign app overlay of app content over web content. The OS might support an app-defined Z-order of elements within an app, such as proposed in [24]. Furthermore, the VPNService used by WIREframe, which exists to prevent an app from spoofing the WIREframe service, could use in-app anti-spoofing methods.

We plan to explore this avenue in future work.