HybridGuard: A Principal-based Permission and Fine-Grained Policy Enforcement Framework for Web-based Mobile Applications

Phu H. Phung[1], Abhinav Mohanty[2], Rahul Rachapalli[2], and Meera Sridhar[2]
[1] Intelligent Systems Security Lab
Department of Computer Science, University of Dayton, Dayton, OH, USA
http://academic.udayton.edu/PhuPhung/
[2] Department of Software and Information Systems, UNC Charlotte, Charlotte, NC, USA
{amohant1,rrachapa,msridhar}@uncc.edu

***Abstract*—Web-based or *hybrid* mobile applications (apps) are widely used and supported by various modern hybrid app development frameworks. In this architecture, any JavaScript code, local or remote, can access available APIs, including JavaScript bridges provided by the hybrid framework, to access device resources. This JavaScript inclusion capability is dangerous, since there is no mechanism to determine the origin of the code to control access, and any JavaScript code running in the mobile app can access the device resources through the exposed APIs. Previous solutions are either limited to a particular platform (e.g., Android) or a specific hybrid framework (e.g., Cordova) or only protect the device resources and disregard the sensitive elements in the web environment. Moreover, most of the solutions require the modification of the base platform.**

In this paper, we present HybridGuard, a novel policy enforcement framework that can enforce principal-based, stateful policies, on multiple origins *without* modifying the hybrid frameworks or mobile platforms. In HybridGuard, hybrid app developers can specify principal-based permissions, and define fine-grained, and stateful policies that can mitigate a significant class of attacks caused by potentially malicious JavaScript code included from third-party domains, including ads running inside the app. HybridGuard also provides a mechanism and policy patterns for app developers to specify fine-grained policies for multiple principals. HybridGuard is implemented in JavaScript; therefore, it can be easily adapted for other hybrid frameworks or mobile platforms without modification of these frameworks or platforms. We present attack scenarios and report experimental results to demonstrate how HybridGuard can thwart attacks against hybrid mobile apps.

I. Introduction

Web-based mobile application (app) development is a technology to develop mobile apps using the web platform (i.e., HTML and JavaScript). Different from regular mobile app development–where the code is written in a native programming language such as Java for Android–the core business code of web-based mobile apps is written in JavaScript and HTML as webpages. The HTML and JavaScript code is then included into a regular mobile app for a particular platform automatically by a middle-tier web-based development framework such as Cordova (https://cordova.apache.org/). Since these web-based mobile apps contain web code (written by developers) and code in a native programming language (normally generated by a framework), they are also known as *hybrid* mobile apps.

Hybrid technology allows mobile apps to be *write-once-run-everywhere* [23], saving time and human resources required for the development process by generating the same app for different mobile platforms (e.g., Android and iOS) with a single development process. *Write-once-run-everywhere* substantially increases revenue for developers because a hybrid app can reach more users in different platforms with less effort. Such advantages render web-based mobile app development a pervasive trend in the mobile industry. In April 2015, a survey of 178 IT organizations revealed that 65% organizations prefer to use hybrid frameworks for mobile app development [28]. Besides Cordova as the most popular one, there are more than seventy such hybrid app frameworks [6] released in the last few years such as Ionic (http://ionicframework.com), Onsen (http://onsen.io), Intel-XDX (https://software.intel.com/en-us/intel-xdk), and Sencha Touch (https://www.sencha.com/products/touch), to name just a few. Only a few months back in July 2016, Facebook released React Native [9], a JavaScript library that is anticipated to be the future of hybrid app development due to its high performance, ability to provide a highly responsive and fluid-like UI, and third-party plug-in compatibility.

Unfortunately, the advent of web-based mobile application technology only exacerbates the security problems of mobile apps. A recent large-scale study of nearly one million web-based mobile apps revealed that 28% of them (i.e., about 280,000 apps) have at least one vulnerability that attackers can exploit to launch serious cyber-attacks [27]. Various other studies have demonstrated that web-based mobile apps expose the device to web-based vulnerabilities, which do not exist in typical native mobile apps [11], [12], [18], [19]. For example, hybrid mobile apps are vulnerable to cross-site scripting attacks that can access the device resources and steal user's sensitive information [18].

In hybrid mobile apps, the core business code is implemented in JavaScript and executed within an embedded browser. There are JavaScript "bridge" APIs, typically provided by hybrid frameworks, that allow the business JavaScript code to interact with the native code to access the device resources and functionalities such as geolocation, contact lists, SMS, and others. The permission model in mobile platforms (i.e., grant all or nothing) is too coarse-grained to prevent the misuse of the JavaScript bridges. For example, let us consider a benign, free hybrid app that has been downloaded to a device and granted permission to access the device resources such as geolocation and SMS. As a norm, this free app includes advertisements (ads) to gain revenue as in the common ad revenue business model [34]. These ads are usually written in JavaScript and other web technologies, and therefore have access to all the available JavaScript bridges [39]. Some hybrid frameworks such as Cordova, provide a whitelist mechanism to prevent access to and from untrusted domains. Content Security Policy (CSP) [26] can be used in hybrid apps to prevent code injection attacks and untrusted external JavaScript code execution. However, these mechanisms do not apply in the ad revenue model since the developer must allow the ad code to run by whitelisting its domain. Usually, the ads are checked by a very thorough screening process to ensure that they are safe, however, the process is not air-tight. In practice, there have been many incidents in the past where malicious ads have made it through the screening process to the users' smartphones [38]. There have also been cases where the entire ad network was compromised, and malicious ads were supplied to the user's devices [2], [17], [8]. By default, any JavaScript code running inside a hybrid app has access to these JavaScript bridges if the user has granted the required permissions [3]. As there is no mechanism to prevent JavaScript code from accessing the JavaScript bridges [39], malicious ads (JavaScript) can e.g., steal users' sensitive information such as geolocation, contact lists transfer the data through phone channels such as SMS. Adapting existing JavaScript security solutions (e.g., [31], [21], [1], [37], [30], [20], [40], [24], [32], [16]) is not straightforward because JavaScript bridges are different from the regular JavaScript APIs in the web, and there are many phone-related channels such as SMS that cannot be captured by regular web security solutions or CSP (We will discuss in more detail about this in Section VIII).

Security issues of hybrid mobile apps recently get more attention from the research community. However, these research studies in hybrid mobile app security face at least one of the following significant limitations. Some solutions focus on a specific platform (e.g., `WebView` in Android), therefore they do not work on other platforms [39], [13]. A few other works provide platform-dependent solutions and focus on a specific hybrid framework such as PhoneGap by modifying them [12], [19]. Some of the solutions provide principal-based access control for device resources [13], however, they do not protect users' sensitive information stored in the web environment. PhoneWrap [11] can protect both device resources and users' information but cannot enforce different policies for multiple principals or origins.

In this work, we propose a robust, and extensible policy enforcement framework for hybrid mobile apps that fills the gaps in the literature mentioned above. Using our HybridGuard framework, developers can define principal-based fine-grained permissions for different origins and can enforce fine-grained, stateful policies to prevent potential cyber-attacks as discussed earlier. The policies will be defined at the development stage in the web code and can be enforced at runtime. By injecting policy code before deployment, our techniques can precisely monitor all web code to ensure its security. The main contributions of our work include:

- We develop a robust framework for hybrid mobile app developers to specify and enforce useful security policies to protect the users from potential cyber-attacks.
- We develop a novel principal-based permission access control and fine-grained security policy specification for hybrid mobile apps.
- We provide a wide-range of security policy patterns that can be enforced in hybrid mobile apps to prevent real-world attacks.
- We report a small-scale experimental evaluation of our proposed framework on Android and iOS platforms.

The rest of this paper is organized as follows. The next section presents the background for our work including the hybrid mobile app architecture. In Section III, we identify the threat model and present motivating attack scenarios. Section IV introduces the overview of our technical approach. The implementation of our framework is detailed in Section V, where we also analyze the security properties our proposed framework. Section VI classifies different policies patterns. We report our experimental results in Section VII. Related work is discussed in Section VIII and conclusion and discussion of our work is presented in Section IX.

II. Background

A. Hybrid Mobile Applications

A hybrid mobile app is built using web technologies – developers uses a hybrid app framework to build the app once using technologies such as HTML5, CSS and JavaScript, and the framework provides mechanisms to instantly port to various mobile platforms such as iOS, Android, Windows Phone and others. The core business code of the app is written in JavaScript and housed in a native container. The framework supplies the app with necessary native code for API access to the device resources. The ability to create a "write-once-run-anywhere" app, with the capability of device access provides the mobile developer unprecedented simplicity and flexibility, simultaneously affording him the superior functionality and user experience of native apps. An attractive advantage is that the developer need not familiarize himself with tools or the skill-set required for developing for a specific native platform [14].

Fig. 1 describes the basic architecture of a hybrid mobile app, in the style of recent related work [13]. The embedded web browser is used as a container to render web content, which can include local web code, remote web code located on the app's web server, and third-party web code such as advertisements or other external JavaScript code. The mobile platform provides

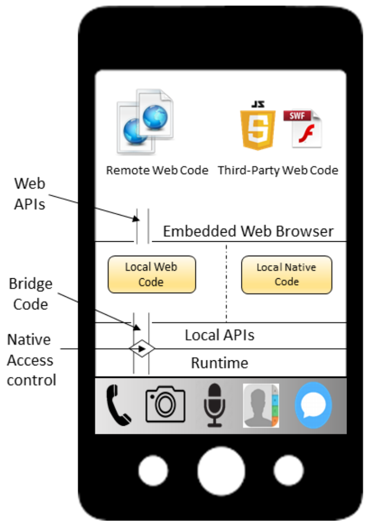

Fig. 1: Architecture of web-embedded mobile apps.

an infrastructure for the web content inside the app's embedded browser to communicate with device resources, such as the microphone, camera, contact list, etc., through *bridge* code, which consists of *web APIs* and *native APIs*.

The embedded browser provides web APIs (HTML5 and JavaScript APIs) for the app's web content to communicate with the native APIs of the device. For example, `navigator.geolocation` is a read-only property of the web API `navigator` which returns a `Geolocation` object that gives the web content access to the device's location. The web APIs also allow the web code to manipulate the web content objects – Document Object Model (DOM).

Native APIs are provided by the underlying operating system and allow the app's web content, via the web APIs, to access the device resources. In Android, native APIs are in Java, and in iOS they are in Objective C [15]. For instance, `android.preference` is a native API that provides classes that manage application preferences and implement the preferences UI. The `android.preference.CheckBoxPreference` class is used to provide checkbox widget functionality in an app.

B. Security in Hybrid Mobile Applications

Apache Cordova, one of the most widely used hybrid mobile app frameworks, has certain security mechanisms in place to reduce the attack surface.

1) Domain Whitelisting: Cordova provides this security mechanism where developers can configure a security policy to define which external domains can be accessed by the hybrid app. The default setting allows access to any external domain [5]. However, this whitelist mechanism is not applicable for third-party JavaScript code within a whitelisted webpage, e.g., the local webpage.

2) Iframes: If content is served in an *iframe* from a whitelisted domain, that domain will have access to the Cordova bridges. Therefore, if the app developer whitelists a third-party advertising network and serves those ads through an *iframe*, it is possible that a malicious ad could break out of the iframe and perform malicious actions. Cordova advises not to use iFrames for this reason [4].

3) Content Security Policy (CSP): CSP support is a native browser capability that allows a developer to control exactly what content his app can access and at a very granular level. CSP is applied at page level for hybrid mobile apps by using a meta tag. By default, applying a CSP disables both `eval()` and inline scripts. Only domains defined in the CSP meta tag can be used to load scripts from or communicating back from the app. However, as CSP can only allow or disallow a certain domain, this makes it coarse-grained. If the app developer wants to load a third-party JavaScript, he has to whitelist this third-party domain using the CSP meta but there is no way for the developer to control the behavior of the third-party code, which can be potential malicious.

III. Threat Model and Running Attack Scenarios

A. Threat Model

In this work, we consider the scenarios that hybrid mobile apps are legitimate and trusted by the users. We assume that code injection attacks in hybrid apps [18] are prevented by the Content Security Policy (CSP) [26] mechanism. The in-scope threats we consider come from third-party JavaScript code that the developers include in the hybrid apps. To function correctly, these included scripts must be allowed in CSP by the developers. However, once included in the apps, third-party JavaScript code has the same privileges as the first-party code and the developers have no mechanism to control their behaviors. In this model, the third-party JavaScript could possibly be (1) benign but may be under control of an attacker through e.g., a SQL injection or network attack on the third-party server, or (2) malicious by intentions e.g., by luring the developers to use its appealing functionalities. In the next subsection, we list several motivating attack scenarios from this threat model.

B. Running Attack Scenarios

1) Misuse of mobile device resources: Consider a hybrid mobile app that requires access to device resources such as geolocation, SMS, Camera, Gallery, and File Storage. By default, after a user grants the required permissions (could be at the installation time or on the fly), any JavaScript code running inside the application has access to these device resources [3]. If a third-party domain, which has been whitelisted by the developer, is infected with malicious code controlled by an attacker, the malicious code can access all the device resources that the app has access to. For example, the malicious code can access the device's photo library or access the device's camera to capture the user's actions live. In addition, the malicious code can also manipulate the DOM of the hosting page including the creation of new elements or the modification of existing elements.

2) Sensitive information leakage: The malicious JavaScript running inside a hybrid app can also read the user's sensitive information including device resources such as contact lists, file storage and personal information such as social security number that may be available on the hosting page. Although the Content Security Policy mechanism disallows information to be sent to any external domain not in the whitelist, any app with access to APIs such as SMS and Email can use these channels to leak the stolen sensitive information.

3) UI attacks: These attacks are known as clickjacking in the web or touchjacking on smartphones [29]. Leveraging the ability to create new elements in the hosting page as mentioned previously, malicious JavaScript code in a hybrid app can launch these attack by creating an invisible interface e.g., an iframe on top of the app interface. When the user touch on the app, he actually touch on the invisible interface, which can lead to download a malicious application to the phone.

IV. Overview of the Proposed Approach

A. Overview

Our enforcement mechanism allows hybrid mobile app developers to include each script code written in a `*.js` file under a principal, which is the core for policy definition and enforcement. As we mentioned in the attack scenarios, third-party code is potentially malicious. Instead of including third-party code directly into the main webpage of the hybrid app, we provide an interface to load the third-party code under a principal named by the developer for tracking purpose. The developer can also load local JavaScript files under a named principal so that he can define fine-grained policies for that trusted principal as well. We manage the principal at runtime so that all API calls from a principal can be tracked.

The second part of our framework is the monitor to control the API execution. Our monitor will intercept security-relevant API calls including access to the device resources and the DOM. By deploying the monitor, any calls to the intercepted APIs will be marked by a principal and checked by the monitor code. The monitor code will invoke the policy engine to decide whether to grant or deny the call based on the defined policies.

B. Inclusion of JavaScript Code

The conventional JavaScript code inclusion is through `<script>` tag, which causes security issues as discussed previously. Our goal is to execute each JavaScript program (in a .js file) under a principal so that the code execution will be marked with that principal at runtime. As a result, we can differentiate which principal calls a certain API. To achieve this, we provide a new API call to load and execute a JavaScript program file. This API allows the developer to assign a principal to a JavaScript program and execute it without using conventional JavaScript inclusion. At runtime, any code execution from this program is marked with the assigned principal so that the policy engine can enforce principal-based policies.

C. Principal Management and Tracking

JavaScript code is loaded and executed in sequence, in order of appearance and "run-to-completion" [10]. However, at runtime, the code can be generated and executed on the fly. This code includes the one generated dynamically or the code embedded into event handlers. In web-based mobile app scenarios, there might be a trusted principal (local code) and multiple third-party principals. Therefore, we must manage the principals and track them when the context is changed due to dynamic code generation and event triggers. We use a local principal stack to keep track of the various principals at runtime as the code is in the execution sequence. Whenever a JavaScript code is executed by our interface, its principal is pushed to the stack. When the code terminates, the principal is popped from the stack. For dynamic code generation and event handlers, we catch them explicitly and execute them under the same principal that generates the code. When the dynamic code or event handlers run, we are still able to attribute their principal to enforce policy for the corresponding principal.

D. API Interception

HybridGuard enforces security policies based on principal and its calls to JavaScript APIs in web-based mobile apps. These APIs include device resource accesses and DOM operations. We intercept these API calls by wrapping them and checking the policy to determine if a call is allowed or not. Our interception and policy code is implemented in one single JavaScript file and included in a `<script>` tag in the main HTML right after the JavaScript "bridge" APIs file (e.g., `cordova.js` for Cordova framework). When executed, it mediates all guarded JavaScript "bridge" APIs and DOM APIs and then loads the required JavaScript code including the local code as well as any remote, and third-party code that the developer intends to include in the app. Loading the JS code through our interface guarantees that this code cannot access any resource via the original APIs but via our mediated APIs so that we can control the execution based on defined policies.

E. Principal-based Permission and Fine-grained Security Policies

Our goal is to develop an enforcement framework that allows developers to specify rules on how JavaScript code from different principals and parties interact with device resources and user's sensitive information. To this end, our policy specification supports the following.

a) Principal-based Permission: We extended the permission model in mobile architecture with principals. For each resource access or action, the developer can define which principal can be granted the access. We support not only allowed/denied for each principal per resource but also provide access qualifiers such as *read, write*, and *create*. We also support policy input such as *whitelist and bound* in this permission specification, which will be used to check fine-grained policies.

b) Stafeful and Fine-grained Security Policies: Some policies cannot be expressed in a static permission or access control rules. For example, to prevent potential information leakage to protect the users, the developer might want to enforce a policy that disallows SMS send if an untrusted principal has accessed the geolocation API. Our enforcement mechanism allows the developer to define local security states based on principals and to update the state so that such stateful and fine-grained policies can be enforced at runtime.

c) Custom Policies: Since our framework is written in JavaScript, the developer can express any custom policies that can not be generalized in rules. For example, to prevent a touchjacking attack mentioned in the attack scenarios (c.f. Section III), the developer needs to intercept the UI actions and check if it violates certain rules.

V. Implementation

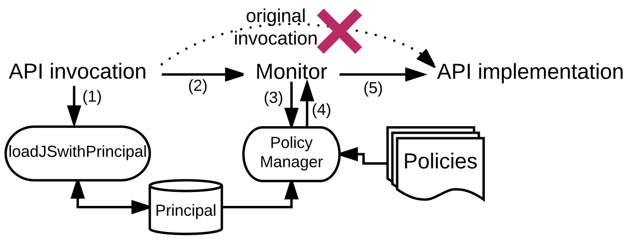

Fig. 2: Conceptual architecture of our proposed framework.

The overview of our framework is illustrated in Fig. 2. A JavaScript program written in a .js file will be loaded under a principal by our new API so that any invocation from the code is marked by that principal at runtime. Any invocation to the guarded APIs will be forwarded to the monitor, which consults the policy manager to check permission and then security policies. If there is no violation, the invocation is allowed and passed to the original API. In this section, we describe the technical details of our implementation of the approach introduced in the previous section.

A. Custom Script Execution with Principal

As discussed, the origin of JavaScript code in hybrid mobile apps is not propagated therefore the app developer cannot enforce policy rules based on the real origin of the invocation [39]. To solve this issue, we introduce and implement a new JavaScript API `loadJSwithPrincipal(p, url)` to replace the conventional script inclusion. The app developer can use this API to load and execute a JavaScript file, local or remote, described in the url argument under a principal `p`. For example, instead using `<script src="http://example.com/ad.js"></script>` to load the external JavaScript from `example.com`, the app developer can use the `loadJSwithPrincipal(..)` to load the code under a principal "`example.com`" as `loadJSwithPrincipal("example.com", "http://example.com/ad.js");`.

We adapted a previous approach [31] to implement this `loadJSwithPrincipal` API. Different from the previous approach, we use CORS (Cross-Origin Resource Sharing) (https://www.w3.org/TR/cors/) to retrieve the content of JavaScript file in a string. Because CORS allows cross-domain communication based on the `XMLHttpRequest` object, we can retrieve either local files or any cross-domain remote file in the same way. We then create a new `Function` object with the retrieved JavaScript content. We then push the assigned principal `p` to a local protected stack (simply implemented as an array), execute the function, and pop the stack after the execution is complete.

B. JavaScript APIs Mediation

We provide a mechanism for the developer to monitor JavaScript APIs including the DOM/HTML5 APIs and JavaScript bridge APIs. We implement this monitoring mechanism by intercepting corresponding API calls so that any invocation to these APIs will be dispatched to the monitor. The monitor will invoke the policy engine to determine whether to allow the invocation. This mechanism is inherited from prior work [32], and depicted in Fig. 2. We note that we have advanced the previous work by implementing mediation for JavaScript bridge APIs and principal-based permission access control, which does not exist in the state-of-the-art JavaScript security solutions.

One challenge in this implementation is the complete mediation of the interception to ensure that JavaScript code can not access the guarded API directly but through the monitor. For DOM/HTML5 APIs, this can be archived by capturing all possible aliases of the guarded API through its prototype inheritance chain in the monitor. There have been several known vulnerabilities in JavaScript that can be exploited in JavaScript interception implementation [25], [22]. We apply the secure wrapper implementations in the literature [22] to ensure that our monitor implementation is tamper-proof from potentially malicious code.

For JavaScript bridge APIs, there might be several different APIs provided by various plugins (JavaScript libraries) to access a device resource. As the app developer includes plugins to her app, she knows the specific APIs to intercept and enforce policies. Each JavaScript bridge API typically uses an internal function call to interact with the native API. For example, in Cordova, `exec` is the internal function to interact with Java API. To ensure that JavaScript code loaded by our framework cannot interact the native APIs directly, we also need to intercept this internal function.

Principal Propagation in Event Handlers and Dynamic Code Generation: Like native mobile apps, hybrid ones are heavily relied on events such as user touch to trigger its computation. In our framework, we capture and intercept these event channels such as `addEventListener`, `attachEvent` to wrap the handler functions so that when the event is fired, e.g., a button is touched, the handler function is executed under the same principal of the parent code so that it will be enforced the same policy for that principal. This approach is illustrated in Listing 1

The same approach is applied for code generation on the fly through DOM APIs such as `document.write`, `Node.insertBefore(..)`. As inline JavaScript code in HTML is not allowed by default CSP, we only need to take care new script node inclusions in the same way of events to ensure that any code generated at runtime will be executed as the same principal as the script created it.

C. Policy Management and Enforcement

As illustrated in Fig. 2, an invocation to a guarded API will be dispatched together with its principal to the corresponding monitor. The monitor then consults the policy manager; based

```
var eventguard = function(args, proceed) {
      var principal = getTopofPrincipalStack();
      var listener = args[1]; //the handler function
      args[1] = function(){
          //wrap the handler function to be executed
        //with the same principal
          return execWithPrincipal(principal,listener);
      };
      return proceed();//register the event handler
    };
intercept(Element.prototype, 'addEventListener', eventguard);
intercept(Node.prototype, 'addEventListener', eventguard);
```

Listing 1: Principal Tracking for event handler

```
{"resources": [{
  "name": "contacts",
  "permissions": [{
   "principal": "local",
   "read": "true",
   "write": "true"
  },
  {
   "principal": "trusted.com",
   "read": "true"
  }]
//...
}
```

Listing 2: A principal-based permission example

on policy definition, the policy manager will decide whether to proceed the invocation. As briefly outlined in the previous section, our framework supports principal-based permission and stateful policies. We design and implement the policy specification for HybridGuard as follows.

1) Principal-based Permission: We use JavaScript Object Notation (JSON) to specify principal-based permission for the device resource access (including DOM and JavaScript bridge APIs) by any JavaScript code running inside the app. The device resources are specified as an array of objects inside the JSON file, and each device resource object has an array of permission objects of its own. The permissions to access the device resources are defined by a principal. For each resource, the app developer can specify which principal can be allowed to read or write. For instance, Listing 2 illustrates an example of principal-based permission that allows the local code (loaded with principal "`local`") to read and write on the contact resource, while allows JavaScript code from "`trusted.com`" read only permission. JavaScript code loaded with other principals is denied access to this resource by default in this example.

This JSON specification can be defined and stored in a local variable within the monitor code, however, to separate policy definition from the code, we store it in a local JSON file and load it using `XMLHttpRequest` to perform principal-based permission check for the policy manager.

2) Custom and Fine-grained Security Policy Enforcement: Principal-based permission can enforce if a principal is allowed to access a resource; however, it cannot capture and prevent potential malicious actions such as sensitive information leakage or UI attacks as we discussed in the motivating attack examples. In addition to the principal-based permission check, our framework also allows the developer to define custom and fine-grained policies such as whitelist specification, stateful, and history-based policies. These policies can also be generalized in a specification; however, we leave this for future work. In this framework, these custom policies can be defined in JavaScript code. For example, to prevent a potential information leakage, the developer can define a policy that "after a principal read the contact list (assume that the principal is allowed to read the contact list in principal-based permission), it is not allowed to send any SMS". This policy is illustrated in Listing 3. We note that this policy is also *principal-based*: the principal violating the aforementioned example policy is denied to send SMS, but other principals such the first-party code can still be allowed to send SMS.

D. Security Analysis

As discussed earlier, potential code injections and information leakage attacks by the web channels can be eliminated by the standard Content Security Policy (CSP) in hybrid mobile apps. Our HybridGuard framework provides an extra layer of protection on JavaScript code that is allowed by CSP. As required by default CSP, each JavaScript code must be defined in a .js file, either first-party or third-party code. HybridGuard provides a new JavaScript API to obtain the content of these .js files and executed them under a principal. This requires the HybridGuard's code to run before other first-party or third-party code in the app so that HybridGuard has the highest priority to control the behaviors of the loaded code. As described in the implementation, the HybridGuard's code and security states are protected within an anonymous function, which is inaccessible from outside code outside. Access to `JSON` policy specification files is prohibited from unauthorized principals, enforced by the monitor. Therefore, the integrity of HybridGuard is guaranteed. Adapted the known techniques from prior work [22], HybridGuard ensures the *complete mediation* of JavaScript web APIs by systematically explores and mediates all their possible aliases and channels generating JavaScript code on the fly. For JavaScript bridge APIs provided by hybrid frameworks, we have to manually identify the possible channels for each API to ensure it is completed wrapped. As HybridGuard can control the behaviors of the loaded code, any unauthorized access can be detected and prevented.

VI. Fine-Grained Security Policies

As discussed earlier, in addition to principal-based permission specification, our framework allows hybrid app developers to define more fine-grained security policies. Implemented as a reference monitor, our framework supports fine-grained security policies satisfying safety property of execution, i.e., preventing bad things happen. The app developer knows the functionality of the app, which resources she will request permission from the user as well as confidential information in the webpage of the hybrid app. When including third-party code, the developer can, therefore, define permission for each party through a principal. In this section, we present some useful policy patterns that the hybrid app developer can leverage to protect the end-users.

```
var contact_read_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"contacts", "read"))
        return; //no permission for this principal
    toggle(contact_read);// update the contact read history
    if(!bound_check(p, "contact", "read") return;
    return proceed();//allow the invocation
  };
var sms_send_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"sms", "send"))
        return; //no permission for this principal
    if(contact_read) return;
    if(!bound_check(p, "sms", "send") return;
    if (!whitelist_check(p, "sms", "send", args[1])) return;
    return proceed();//allow the invocation
  };
intercept(sms, 'send', sms_send_policy);
intercept(navigator.contacts, 'find', contact_read_policy);
```

Listing 3: Example of "no SMS send after reading contact list"

Resource bounds Policy: In general mobile apps, it is alarming that apps overuse or abuse the resources by invoking it a significant number of times [11]. In some scenarios, just disallowing access to a resource is not an adequate policy. For example, when including an advertisement code, the developer needs to allow the advertisement to access the `geolocation` resource; disallowing this access might break the functionality of the ad code. However, the developer might want to limit the number of times of access to the `geolocation` resource to prevent potential information leakage attacks as malicious ads can periodically read and send out geolocation information.

In HybridGuard, the developer can specify a bound limit for each resource per principal in each resource access action policy. We support the app developers by providing a `bound_check(principal, resource, action)` function to update the history of the access and check if it reaches the bound limit. The use of this function is illustrated in the policies in Listing 3.

Whitelist Policies: In some scenarios, a principal is allowed to invoke an API with parameters. For example, to send a text message, the code need to call `sms.send` with the number to be sent together with other parameters. The developer might want to limit that principal to send to a limited list of receivers. This might prevent a user's private information to be leaked to unwanted recipients. In another example, the developer might allow a third-party principal interact and modify only a particular element in the DOM. To support this, we have implemented a `whitelist_check(principal, resource, action, arg)` to check if the argument is defined in the corresponding whitelist for the action and the principal. The app developer can invoke this check in the policy for a certain API call when intercepting that API. An example of using this function in policies is illustrated in Listing 3.

History-based Policies: A common attack in e.g., malicious advertisements is to read sensitive user data and send it to the attacker through different channels such as image source. Although CSP policy can prevent some of these channels so that the leakage can be limited, there are other channels specific to a mobile device that are not captured by CSP such as SMS, and email. The developer can prevent this potential information leakage by monitoring the access to sensitive information and preventing some certain APIs that is not captured by CSP. For example, the developer can define a policy "no SMS sending after contact list is read" by intercepting the contact read action and toggle the contact read flag, which can be checked in the policy for SMS send: if the flag is toggled, the SMS send action is disabled. This whole policy is defined in Listing 3.

Custom Policies: There are several potential malicious behaviors of third-party JavaScript code such as manipulating the DOM and create UI attacks such as touchjacking (e.g., by creating an invisible iframe) or open a webpage to launch a phishing attack. In our framework, in addition to the supported policies presented above, the app developer can implement any custom policies in JavaScript when intercepting HTML5/DOM APIs and JavaScript bridge APIs. In the touchjacking example, the developer can enforce a policy that disables the creation of an invisible `iframe`.

VII. Experimental Results

In this section, we present the results of our experimental evaluation. We have developed the core framework within a JavaScript program inside an anonymous function `(function(){ /* code */})();` so that its code and security states can be protected. The interception implementation and policy engine are combined within this anonymous function comprising of ~800 lines of JavaScript code. To deploy our framework in a hybrid app, the developer just need to copy this library together with the JSON permission specification file to the `www` folder of the app, then include it (use e.g., `<script src="HybridGuard.js"></script>`) to the main HTML page, *e.g.,* `index.html` right after the core JavaScript library of the hybrid app (e.g., `cordova.js` in the case of Cordova app). As introduced earlier, to include a JavaScript program (in a file, local or remote) the developer can use our API `loadJSwithPrincipal(principal,url);` to load and execute code under a principal, instead of including these programs as the conventional way of `<script>` tag. This loading code can be implemented in a separated JavaScript file after `<script src="HybridGuard.js"></script>`) or can be placed at the end of `"HybridGuard.js"` file outside the anonymous function. After assigning principals for different JavaScript files, the developer can edit the policy JSON specification file to define fine-grained permission for each principal. To evaluate the effectiveness of HybridGuard, we have tested it within a hybrid app developed by ourselves, and have deployed it to several real-world Android apps from Google Play.

A. Testing on self-developed hybrid mobile app

We use Cordova framework (version 5.3.3) to develop the testing app. We include several resource plugins listed in Table I such as SMS, email, contacts, camera, geolocation, accelerometer, File System and develop their functionality in local JavaScript files, and load them with "local" principal using `loadJSwithPrincipal("local",<js-file>);`. We also host similar JavaScript files remotely and load them with "remote" principal using `loadJSwithPrincipal("remote",<remote-js>);`. We specify the principal permission in the JSON file to allow/disallow some access to the resource by a principal. We have performed several minor modifications in the policy code to make it consistent with the plugins and policies. All policies introduced in the previous section have been implemented. We use Cordova to build the app for both Android and iOS platforms. For Android, we deploy the app directly to real devices Nexus 5X and Nexus 6P running on the Android 7.1.1 (Nougat). For iOS, we use Xcode (version 7.2.1) to build and deploy the app to an iPhone 6s Plus iOS 9.2 simulator. We turn on the debug messages so that we can observe the principal propagation is tracked correctly. The permissions to the device resources are checked at runtime correctly based on principal. Fine-grained policies such as information flow and history based policies are soundly enforced. We note that Cordova has been used for our testing; however, as HybridGuard is developed in JavaScript, it can be easily adapted and applied for other hybrid mobile frameworks with some minor modifications in the enforcement and policy code.

TABLE I: List of Policies Enforced on Plugins

Resource	PlugIn and Resource object	Method	Policy Enforced
Files	cordova-plugin-file Object : window.requestFileSystem	requestFileSystem	Whitelist History-based Policy
Camera	cordova-plugin-camera Object : navigator.camera	getPicture	No Send after read
Contacts	cordova-plugin-contacts Object : navigator.contacts	find	Whitelist History-based Policy Resource bounds policy
Accelerometer	cordova-plugin-device-motion Object : navigator.accelerometer	getCurrentAcceleration watchAcceleration	Whitelist Enforcement
SMS	cordova-sms-plugin Object : sms	send	Whitelist History-based Policy Resource bounds policy
Geo Location	cordova-plugin-geolocation Object :navigator.geolocation	getCurrentPosition watchPosition	History-based Policy Resource bounds policy
Video Recording	cordova-plugin-media-capture Object : navigator.device.capture	captureVideo captureImage	Whitelist
Secure Storage	cordova-plugin-secure-storage Object : cordova.plugins.SecureStorage	SecureStorage SecureStorage.get SecureStorage.set	History-based Policy Whitelist

B. Testing on real-world Android hybrid apps

We have performed a small-scale evaluation on real-world Android hybrid apps by manually download some Android apps marked as a hybrid app from Google Play using `apkpure.com`. We use a reverse engineering tool `apktool` (https://github.com/iBotPeaches/Apktool) to decode resources to nearly original form (use e.g., `apktool decode -f -s apkFile.apk`). We include the framework library, i.e., `HybridGuard.js` and permission JSON file to the `www` folder, and modify the main page to include the library and load the core scripts. Similarly, in the testing app, we also do some minor modification in policy code to adapt the APIs. After the modifying the web in the `www` folder, we rebuild the app using the `apktool` (use e.g., `apktool build modifiedApkFolder/`). The app is then signed using `jarsigner` (use e.g., `jarsigner -verbose -keystore your.keystore modifiedApkFile.apk`) and is installed on the device.

We have downloaded ten hybrid application apks from Google Play through `apkpure.com` and modified them by manually including HybridGuard framework as described above. A few apps that have been tested successfully are Parked Car Locator, Web Ratio, Remote SMS Control, Graded, Fan React, My Car Navigator. These applications access various system resources like Camera, Geo Location, Accelerometer, Contacts, or File System. Policies like limiting the access to resources or send messages and location details only to whitelisted sources, blocking SMS and email sending as soon as a content from a file is read have been enforced. The tested apps will enforceable security policies are listed in Table II.

TABLE II: List of tested hybrid mobile apps

Application Name	Resources Accessed	Policies
Parked Car Locator	Geo Location	Whitelist Enforcement
My Car Navigator	Geo Location Accelerometer	Whitelist Enforcement Resource bounds policy
Fan React	Contacts SMS	Whitelist Enforcement History-based Policy Resource bounds policy
Graded	SMS Contacts File System	Whitelist Enforcement History-based Policy Resource bounds policy
Remote SMS Control	SMS Contacts File System	Resource bounds policy Whitelist Enforcement History-based Policy
Web Ratio	Contacts File System	Whitelist History-based Policy Resource bounds policy

C. Performance

We have not yet tested the performance and overhead of our framework, however, when testing the app, we did not notice any significant delay. Prior work on JavaScript interception have reported that the overhead of these implementations is not significant [31], [37], [30], [24], [32].

VIII. Related Work

A. Third-party JavaScript Isolation

There are various proposals in the literature to protect against malicious third-party JavaScript, e.g., [16], [31], [37], [1], [20], [21], [24], [30], [32], [40]. However, adapting these solutions to hybrid mobile app environment is not a trivial task since there are many phone-related resources and channels are not captured by the existing approaches. Several proposed methods such as [1], [31], [30] could be applied but required signification modifications to capture these phone resources and channels. Furthermore, none of these solutions provided principal-based permission for mobile apps as we propose in HybridGuard. JaTE [37] uses `Proxy` in ECMAScript 6 to isolate third-party JavaScript with principals, however, `Proxy` has not yet supported in embedded browsers in mobile platforms (c.f. https://kangax.github.io/compat-table/es6/, March, 2017). Approaches like Adsafe [16] can be applicable, however, it must be extended with JavaScript bridge APIs and it requires third-party JavaScript written in their JavaScript subset. In contract, HybridGuard allows the full set of JavaScript and bridge APIs provided by frameworks. ConScript [24] requires browsers to be modified to enforce security policies. In this case, the base mobile platform must be also modified; therefore it limits the deployment of the protection. HybridGuard does not require the modification of browsers, hybrid frameworks, or the base platforms.

Adjail [20] and Webjail [40] use iframes to isolate third-party content and provide a mechanism for cross-platform interaction. However, these works cannot project against attacks for JavaScript bridge APIs in hybrid apps because they are accessible for any JavaScript code allowed to load in a hybrid app.

B. Hybrid Mobile Application Security

The closest related work to HybridGuard is PhoneWrap [11], which introduced a fine-grained ticket-based policy enforcement into web-based mobile apps to control a bounded number of accesses for each resource based on the user's interaction with the app. Resource accesses through JavaScript interfaces are wrapped by a library, inspired by "`self-protecting JavaScript`" approach [32]. However, PhoneWrap does not investigate a multi-party scenario in web-based apps, and thus cannot define and enforce separate policies for different origins as we proposed in this work. POWERGATE [13] is an access-control mechanism for Web-based system applications where the developers can define principal-based access control policies similar to our work. However, in contrast to the policies implemented by POWERGATE which simply "allow or disallow" a particular native object for each principal such as "local code", "third-party remote code", HybridGuard can enforce stateful policies for multiple principals on each native object. In addition, the implementation of POWERGATE requires the modification of the base system e.g., Firefox OS.

In [19], a fine-grained access control mechanism for web-based mobile apps in Android, using *frame*-level access control has been proposed. A `permissions` attribute is introduced for the `iframe` tag used in `WebView` to specify the device resources the frame has access to. It also introduces an `access` tag that can be added to the Android manifest file to specify resource access permissions to different `origins`. However, this approach is specific to Android as it requires the modification of the Android base system [19]. In contrast, our approach is at the web layer, therefore it can be applicable to any mobile platform without modification.

Another study proposes a context-aware permission control system for web-based mobile apps [36]. This system can enforce information flow policies to prevent the potential data breach in web-based mobile apps. A page-level access control mechanism has also been proposed in the past, which provides a particular page access only to the device resources it requires, to minimize the attack surface [35]. However, the solution is only applicable to multi-page web-based mobile apps.

Draco [39] provides uniform and fine-grained access control for web code running on Android in-app browsers. It provides a declarative policy language for developers to define their own fine-grained access control policies for multiple origins and also provides the Draco runtime system (DRS) to enforce these policies at runtime. However, the implementation the framework modifies the Chromium Android System WebView app.

RestrictedPath [33] enforces access control at both the browser-level and at the system-level. Developers first define intended paths of their apps; RestrictedPath subsequently monitors all invocations made by the apps and determines whether the app deviates from its intended path [33].

Several solutions focus on detecting code-injection attacks in hybrid mobile apps. Jin et al. introduce the possibility of code-injection attacks in hybrid mobile apps through non-web channels peculiar to smartphones. These channels include the camera in the form of a barcode scanner, SMS, Contact List, Calendar, NFC and even Wi-Fi access points [18]. DroidCIA [7] extends the previous work and introduces a new code-injection channel, where a malicious script can be injected by using the HTML5 `textbox` input type along with `document.getElementByID("TagID").value` [7]. Xiao et al. introduce a new type of code-injection attack, where JavaScript code is encoded in a human-unreadable format [41]. The authors use machine learning algorithms to detect vulnerable apps and also suggest an improved access control model that uses a combination of page-based and frame-based techniques. However, such code-injections can be mitigated by the use of CSP and not allowing the execution of inline scripts. Our work focuses on attacks that cannot be mitigated by CSP.

IX. Conclusion

We have presented the design and implementation of HybridGuard, a robust framework to specify and enforce principal-based and fine-grained security policies to guard against attacks in hybrid mobile apps originating from third-party JavaScript. Our enforcement framework is platform independent as it is developed in JavaScript; thus it can be deployed on various mobile platforms and hybrid development frameworks without modifying them. We have demonstrated the implementation of the policy engine and specification of the principal-based and fine-grained policies. We specify a wide range security policies that the app developer can use to mitigate potential attacks. We have conducted experiments to evaluate the framework and policies on real hybrid apps and mobile devices.

Our in-scope threats come from potential malicious third-party JavaScript code in a hybrid app that a developer explicitly includes; therefore, our framework relies on developers on defining security policies. In practice, the app users might be in a right position to define desired security policies to protect themselves. In future work, we also plan to extend the policy system so that the app users can specify their policies on a hybrid app. We also plan to construct a testbed of hybrid apps and an ontology of possible attacks so that we can conduct a large-scale evaluation of real-world hybrid apps and effective security policies.

Acknowledgements

This research was supported in part by The University of Dayton Research Council Seed Grant, the Department of Software and Information Systems at the University of North Carolina Charlotte, and NSF CRII award #1566321.

References

[1] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 1–10, December 2012.

[2] Alex Hern. Spotify hit by 'malvertising' in app. https://www.theguardian.com/technology/2016/oct/06/spotify-hit-by-malvertising-in-app, October 2016. Accessed: 2017-1-13.

[3] Apache Software Foundation. Cordova - Security Guide. https://cordova.apache.org/docs/en/latest/guide/appdev/security/. Accessed: 2017-1-13.

[4] Cordova security guide. https://cordova.apache.org/docs/en/latest/guide/appdev/security/. Accessed: 2017-1-13.

[5] Apache Software Foundation. Whitelist Guide. https://cordova.apache.org/docs/en/latest/guide/appdev/whitelist/. Accessed: 2017-1-13.

[6] S. Arora. 10 best hybrid mobile app UI frameworks: HTML5, CSS and JS. http://noeticforce.com/best-hybrid-mobile-app-ui-frameworks-html5-js-css, September 2015. Accessed: 2016-11-11.

[7] Y. Chen, H. Lee, A. B. Jeng, and T. Wei. DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps. In *Proceedings of the 14th Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, volume 01, pages 1014–1021, 2015.

[8] Dan Goodin. Millions exposed to malvertising that hid attack code in banner pixels. http://arstechnica.com/security/2016/12/millions-exposed-to-malvertising-that-hid-attack-code-in-banner-pixels/, December 2016. Accessed: 2017-1-13.

[9] Facebook Inc. React - A JavaCript library for building user interfaces. https://facebook.github.io/react/. Accessed: 2016-11-11.

[10] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, 6th edition, 2011.

[11] D. Franzen and D. Aspinall. PhoneWrap-injecting the "How Often" into mobile apps. In *The 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*, pages 11–19, 2011.

[12] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[13] M. Georgiev, S. Jana, and V. Shmatikov. Rethinking security of web-based system applications. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pages 366–376, 2015.

[14] Native vs. html5 vs. hybrid apps. http://www.glowtouch.com/native-vs-html5-vs-hybrid-apps/, June 2015.

[15] Google Inc. Android NDK Native APIs. https://developer.android.com/ndk/guides/stable_apis.html. Accessed: 2016-11-11.

[16] Integral Ad Science, Inc. Effectively influence consumers everywhere. https://integralads.com/. Accessed: 2016-11-12.

[17] Jeremy Kirk. Massive Malvertising Campaign Hits MSN, Yahoo. http://www.bankinfosecurity.com/massive-malvertising-campaign-hits-msn-yahoo-a-9583, December 2016. Accessed: 2017-1-13.

[18] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 66–77, 2014.

[19] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for HTML5-based mobile applications in Android. In *Information Security (ISC)*, pages 309–318. Springer, 2015.

[20] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of USENIX Security'10*, pages 24–41, Berkeley, CA, USA, 2010. USENIX Association.

[21] M. T. Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan. SafeScript: JavaScript Transformation for Policy Enforcement. In *Proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec 2013)*, volume 8208 of *Lecture Notes in Computer Science (LNCS)*, pages 67–83. Springer Verlag, October 2013.

[22] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Proceedings of the 15th Nordic Conference in Secure IT Systems (NordSec)*, pages 239–255, October 2010.

[23] D. McCormick. Future of mobile app development is hybrid apps, not native ones, says new survey. http://www.alphasoftware.com/blog/hybrid-mobile-apps-not-native-ones-will-rule-says-new-survey/, 2015. Accessed: 2016-11-11.

[24] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[25] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of 2010 IEEE Symposium on Security and Privacy*, pages 481–496, May 2010.

[26] Mike West and Joseph Medley. Content Security Policy. https://developers.google.com/web/fundamentals/security/csp/. Accessed: 2017-1-13.

[27] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, May 2015.

[28] State of applicaton development report: Mobility, custom apps a priority for 2015. Technical report, April 2015.

[29] OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking. Accessed: 2017-1-13.

[30] P. H. Phung and L. Desmet. A two-tier sandbox architecture for untrusted JavaScript. In *Proceedings of the Workshop on JavaScript Tools (JSTools)*, pages 1–10. ACM, June 2012.

[31] P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 12(4):443–457, July 2015.

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

[33] S. Pooryousef and M. Amini. Fine-grained access control for hybrid mobile applications in Android using restricted paths. In *Proceedings of the 13th International ISC Conference on Information Security and Cryptology (ISCISC)*, 2016.

[34] Sean Butner. How Much in Advertising Revenue Can a Mobile App Generate? http://smallbusiness.chron.com/much-advertising-revenue-can-mobile-app-generate-76855.html. Accessed: 2017-1-13.

[35] M. Shehab and A. AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle (MobileDeli)*, pages 1–8, 2014.

[36] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *Proceedings of the 16th International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 307–327, 2013.

[37] T. Tran, R. Pelizzi, and R. Sekar. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 151–160, New York, NY, USA, 2015. ACM.

[38] TrendMicro. These ads are more than annoying: Android banking malware to watch out for, August 2016. Accessed: 2017-3-15.

[39] G. S. Tuncay, S. Demetriou, and C. A. Gunter. Draco: A system for uniform and fine-grained access control for web code on Android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 104–115, 2016.

[40] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[41] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, and Y. Jiang. Detection and prevention of code injection attacks on HTML5-based apps. In *Proceedings of the 3rd International Conference on Advanced Cloud and Big Data (CBD)*, pages 254–261, 2015.