SPECIAL SECTION ON PRIVACY PRESERVATION FOR LARGE-SCALE USER DATA IN SOCIAL NETWORKS

Received November 20, 2017, accepted January 8, 2018, date of publication January 18, 2018, date of current version March 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2795383

Detecting Malicious Behaviors in JavaScript Applications

JIAN MAO[1], (Member, IEEE), JINGDONG BIAN[1], GUANGDONG BAI[2], RUILONG WANG [1], YUE CHEN[1], YINHAO XIAO[3], AND ZHENKAI LIANG[4], (Member, IEEE)

[1]School of Electronic and Information Engineering, Beihang University, Beijing 100191, China
[2]Cluster of Infocomm Technology, Singapore Institute of Technology , Singapore 138683
[3]Department of Computer Science, The George Washington University, Washington, DC 20052 USA
[4]School of Computing, National University of Singapore, Singapore 117417

Corresponding author: Jian Mao (maojian@buaa.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB0802400, in part by the National Natural Science Foundation of China under Grant 61402029, Grant 61370190, and Grant 61379002, in part by the Singapore Ministry of Education under the National University of Singapore under Grant R-252-000-666-114, and in part by the Funding Project of Shanghai Key Laboratory of Integrated Administration Technologies for Information Security under Grant AGK201708.

ABSTRACT JavaScript applications are widely used in a range of scenarios, including Web applications, mobile applications, and server-side applications. On one hand, due to its excellent cross-platform support, Javascript has become the core technology of social network platforms. On the other hand, the flexibility of the JavaScript language makes such applications prone to attacks that inject malicious behaviors. In this paper, we propose a detection technique to identify malicious behaviors in JavaScript applications. Our method models an application's normal behavior on function activation, which is used as a basis to detect attacks. We prototyped our solution on the popular JavaScript engine V8 and used it to detect attacks on the android system. Our evaluation shows the effectiveness of our approach in detecting injection attacks to JavaScript applications.

INDEX TERMS JavaScript application, hybrid mobile app, behavior anomaly detection.

I. INTRODUCTION

For the excellent cross-platform support, JavaScript applications are widely used to power a wide range of solutions, including web applications, mobile applications and desktop/server applications. As a result, JavaScript is a core technology that supports popular social networking application architectures, covering both the cloud (servers) and the ends (browser and mobile apps). For the convenience of users, such social network applications often require users to provide their geolocation, personal addresses, contacts, etc. On one hand, these applications often offer user-friendly and customized features for users. On the other side, users' privacy is exposed to these applications, and is even potentially revealed to adversaries [1].

In fact, the flexibility of the JavaScript language makes such applications even more prone to attacks that inject malicious behaviors. The existing web vulnerabilities in these application, such as cross-site scripting (XSS) [2], are also carried into JavaScript applications on other platforms, which can lead to security breaches and reveal users' privacy.

In this paper, we focus on JavaScript-based mobile applications, which is also called hybrid mobile apps. Compared to traditional web applications, such JavaScript applications have access to more sensitive channels on mobile devices, such as contacts and messages, allowing malicious code to be hidden in such channels to be injected into hybrid mobile apps [3]–[5]. Moreover, the injected code has much more power in accessing system resources, such as camera and GPS information, than its counterparts in the web.

Researchers have extensively studied code injection on the web platform [6]–[15], the solutions are mainly for the architecture of web applications. They have proposed solutions to prevent code injection attacks in hybrid mobile apps [3], [16], which works mainly by filtering code out of the data input from potential code injection channels of the devices. However, these solutions need the knowledge of potential injection channels, and thus may become ineffective when new injection techniques are developed by attackers.

In JavaScript applications, the anomalous (or foreign) behaviors by injected code break the execution integrity of the victim app, resulting in different behaviors from benign

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

ones in the way they are activated. Based on this observation, the research community has developed solutions to detect anomalous behaviors based on *behavior models* in several platforms, such as x86 programs [17], [18] and web applications [19], [20]. The key of such solutions is to identify the program states under which anomalous behaviors can be distinguished from benign behaviors. In hybrid mobile apps, dangerous behaviors are carried out as call to APIs provided by the hybrid framework such as PhoneGap [21]. These APIs are comparable to system calls to an operating system. Therefore, in our approach, our goal is to identify and prevent malicious call to these APIs. For hybrid Android apps, we observed that the caller-callee relationship of JavaScript functions can provide information to distinguish benign and malicious calls to these APIs. Our approach is based on function activation information of the apps and system events.

A. OUR APPROACH

In this paper, we propose a new approach to detect anomalous behaviors in hybrid Android apps as anomaly in function call behaviors. To intercept function calls of JavaScript in hybrid apps, we dynamically instrument JavaScript code in the JavaScript engine. The instrumented JavaScript reports function-level activity when a hybrid app is executed. In addition, we also extract events from the WebView component to enhance the behavior model. Based on these events, we detect attacks as the deviation between a hybrid app's behavior and its expected behaviors.

We prototyped our solution in the Android system, and evaluated it using real-world hybrid Android apps. Our evaluation shows how it can distinguish behaviors by injected code. Moreover, with the wide deployment of JavaScript-based applications, our solutions can be adopted by JavaScript applications in other domains, such as server-side applications and IoT solutions.

B. CONTRIBUTIONS

In summary, we make the following contributions:

- We propose a new approach for detecting malicious behaviors in hybrid Android apps. The key of our technique is to identify function-level execution information as the basis to distinguish benign and foreign behaviors.
- We develop a dynamic instrumentation technique to extract the function-level run-time information from hybrid application.
- We prototype our approach, and apply it to successfully detect foreign-behaviors in hybrid apps under code injection attacks.

C. ORGANIZATION

The rest of this paper is organised as follows. Section II discusses related work. Section III discusses the background knowledge of hybrid applications and gives an overview of our solution. In Section IV, we illustrate the overall architecture of our approach and present essential algorithms. We describe the implementation details and evaluation results in Section V and Section VI, respectively. Section VII concludes the paper.

II. RELATED WORK

A. HYBRID ANDROID APP SECURITY

Georgiev *et al*. [22] discussed the security flaws of the hybrid application framework and analyzed the vulnerabilities (e.g., fracking), potential attacks introduced by the gap between web application security mechanisms (e.g., SOP) and Android system access control policies. They presented a capability-based solution to prevent malicious code in Android hybrid apps from accessing high privilege, which is platform-independent and compatible with the existing frameworks and embedded browsers without changing the code of hybrid apps nor their business model.

Jin *et al*. [23] studied the code injection attacks introduced in hybrid mobile apps. They discovered a few channels (e.g., 2D barcode reading, WiFi access point scanning, Bluetooth device pairing, etc.) that can be used to conduct code injection attacks through hybrid apps. In their follow up work [16], they identified most potential ways/''bridges'' exposing the system resources to JavaScript without appropriate access control mechanisms, which are dangerous to the emerging threats. They developed tools to detect hybrid apps potentially vulnerable for code injection attacks and proposed a fine-grained access control model to filter out malicious code that can be injected to the vulnerable apps.

B. MALICIOUS BEHAVIOR DETECTION USING BEHAVIOR MODELS

Several solutions pioneered using behavior models to detect malicious system-call behaviors in applications. Forrest *et al*. [24] first proposed the anomaly system call sequence based intrusion detection approach. Sekar *et al*. [17] proposed to model program behavior using a compact finite state automaton (FSA), which models system call sites as states. Feng *et al*. [18] adopted call stack information to the state machine model for better accuracy. In web applications, Guha *et al*. [19] extracted the *non-deterministic request graph* by statically analyzing client-side web applications, which is used to detect anomalous behaviors in Ajax applications. Dong *et al*. [20] used client-side state transactions and communications to servers to build a state machine model for detecting malicious behaviors in web applications. Mao *et al*. [25] extended the solution to detect malicious behaviors in hybrid mobile apps. Such models may not be effective in detecting behaviors injected into the same web interfaces. In our early work [26], we showed that utilize function-call relationships can be used for to detect attacks after it happens. In our approach, we extend the scope of the solution to general JavaScript applications in this paper, and use it to perform online detection of malicious behaviors.

C. CODE INJECTION DEFENSE IN WEB APPLICATIONS

Code injection attacks on web applications, especially cross-site scripting attacks (XSS) [2], that circumvent the same-origin policy [27] can obtain arbitrary access to contents of the vulnerable website. Content Security Policy (CSP) [28] is designed to prevent unauthorized scripts from executing within the applied website. Researchers have employed static program analysis to detect XSS vulnerabilities [6]–[8]. Numerous approaches also prevent XSS attacks by using dynamic tracking techniques to control the use of unsafe data in web applications [9]–[11]. Wassermann and Su [29] and Balzarotti *et al.* [30] proposed solutions to verify the correctness of sanitization functions. Noxes [31] and NoMoXSS [32] developed client-side XSS defenses to ensure confidentiality of sensitive data by analyzing data flow in the browser instead of preventing executing illegitimate scripts. Based on the unique features of reflected XSS attacks by comparing HTTP request parameters and responses, researchers deploy client-side and server-side mechanisms to detect and mitigate these attacks [12]–[14]. DSI [33], Noncespaces [15] and Blueprint [34] preserve the integrity of document structure in the browser to prevent XSS attacks. Meanwhile, a number of researchers focus on confining the behaviors of untrusted scripts by transforming JavaScript code [35]–[38]. Recently, DOM-based XSS attacks emerged, and various techniques are proposed to detect these attacks by taint analysis and mitigate them by auto-patching the vulnerabilities [39]–[42]. Apart from the above defense solutions, a line of approaches focused on utilizing privilege separation on web applications to protect sensitive data from untrusted scripts [43]–[48].

D. PRIVACY LEAKAGE IN ONLINE SOCIAL NETWORKS (OSNs)

Prior research on privacy in online social networks mainly focuses on inferring users' identities and personal information from public information shared in various OSNs [49]–[52]. Zheleva and Getoor [50] devised a classification-based approach to obtain users' sensitive information from their public social relationships and group information. Balduzzi *et al.* [51] leveraged email addresses to determine and linked users across different OSNs. Chaabane *et al.* [52] proposed to infer users' undisclosed (private) attributes using the public attributes of other users sharing similar interests. Leveraging the Latent Dirichlet Allocation generative model, they can extract semantic links between users' unrelated interest names, further postulate and verify the interest-based similarities between users. In particular, they showed that as long as users revealed their music interests, their sensitive attributes, such as gender, age and country-level locations can be revealed with high accuracy.

III. APPROACH OVERVIEW

In this section, we introduce the runtime environment of JavaScript Apps. Based on that, we use a motivating example to demonstrate how the function-activation relationship is effective in differentiating anomalous behaviors.

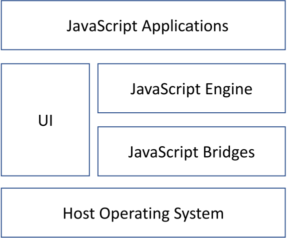


FIGURE 1. The architecture of JavaScript applications.

A. RUNTIME ENVIRONMENT OF JAVASCRIPT APPS

Illustrated in Figure 1, JavaScript applications rely on a JavaScript engine to run its core logic, and on user interface (UI) modules to render its interface. The JavaScript engine interacts with the host operating system through a bridge module. The bridge module provides JavaScript APIs with system resources. It also offers a way to deal with the requirement of particular systems that contain the JavaScript engine in a sandbox. For example, in Android browser, to prevent web pages from accessing system resources, a WebView component is executed within a sandbox. When the WebView is used to execute hybrid mobile apps, it uses a plugin-based middleware framework as the bridge. With this bridge, JavaScript code can invoke native Java code to access system resources.

There are several middleware frameworks that can provide such a bridge, e.g., PhoneGap [21], RhoMobile [53] and Appcelerator [54]. Apps developed based on these cross-platform frameworks are called hybrid apps. The static layouts and dynamic behaviors of these hybrid apps are implemented in Web languages, e.g., HTML, CSS and JavaScript.

B. MOTIVATING EXAMPLE

We use a simplified hybrid app to illustrate how a JavaScript code injection attack occurs, and to understand the intuition that the function-level information is critical for anomalous behavior detection. In this paper, without loss of generality, we base our design on PhoneGap, which is the most popular framework nowadays, and can be used on various mobile platforms such as Android, iOS and Windows Phone. PhoneGap provides 16 plugins as the bridges to enable hybrid mobile apps to access system resources, including file, camera, accelerometer, etc. If new access to local resources is needed, developers can also write their own plugins to extend the functionalities of these frameworks. JavaScript code can access new Android native resources by calling these new plugins. These plugins will directly invoke Java code and serve as bridges between JavaScript code and Android system resources.

The app, called *GroupMessageSender*, contains four main operations, i.e., *Search the contact list*, *Remove the user*

```
function contactSearch() { //Read contact list
  var options = new ContactFindOptions();
  options.filter = "";
  options.multiple = true;
  var fields = ["*"];
  navigator.contacts.find(fields,
    function (contacts) { //Display contact items
        with check boxes
      var ul = document.getElementById("ul");
      for (var i = 0; i < contacts.length; i ++) {
        var contactinfo = document.createElement("p
            ");
        var checkboxInput = document.createElement("
            input");
        checkboxInput.type = "checkbox";
        contactinfo.appendChild(checkboxInput);
        var li = document.createElement("li");
        contactinfo.innerHTML += contacts[i].
            displayName + ":" + contacts[i].
            phoneNumbers[0].value;
        li.appendChild(contactinfo);
        ul.appendChild(li);
      }
    }, function (){}, options
  );
}
```

```
function contactRemove() {
  var options = new ContactFindOptions();
  options.filter = '';
  options.multiple = true;
  var fields = ['*'];
  navigator.contacts.find(fields,
    function remove(contacts) {
      var ul = document.getElementById("ul");
      for (var i = 0; i < ul.getElementsByTagName("
          li").length; i ++) {
        var li = ul.getElementsByTagName("li")[i];
            //Remove the selected items
        if(li.getElementsByTagName("input")[0].
            checked) {
          contacts[i].remove(function () {},
              function () {});
        }
      }
    }, function () {}, options
  );
}
```

FIGURE 2. **The source code of *GroupMessageSender*.**

selected contact items, *Add a contact item*, and *Send SMS message*. It reads the whole contact list and displays contact items with check boxes. Users can either select contact items to send SMS messages to them or delete them from the list. The source code of reading, displaying and removing the contact list is shown in Figure 2.

The functions shown in Figure 2 are invoked as follows in the original app. On user click of buttons to search contacts, the browser invokes `contactSearch()`, which calls browser APIs and the PhoneGap APIs, including `navigator.contacts.find()`. On user click of buttons of removing contacts, the browser invokes `contactRemove()`, which calls browser APIs and PhoneGap APIs, including `navigator.contacts.find()` and `contact[i].remove`. These call relationships *define* the normal behavior of the app.

An attacker can inject malicious code into the name field of a contact item, for example, `<img src="x" onerror="contactRemove()"/>`. When the app reads the mal-formatted contact item, the malicious code will be executed and directly calls the app's JavaScript function `contactRemove()` to delete specific items.

In this attack, the malicious behavior (removing contacts) is caused by the injected code, which is invoked while the user is viewing the contact list. In contrast, the normal behavior to remove contacts should be activated by user clicking on the button, which is designed to remove contacts. Furthermore, the function call stack of this contact remove behavior should only contain the function `contactRemove()` in the normal circumstance. However, in a code injection attack, the same behavior is triggered with a function call `contactSearch()` → `contactRemove()`, which is not available in the original program behavior. These two different behaviors are shown in Figure 3.

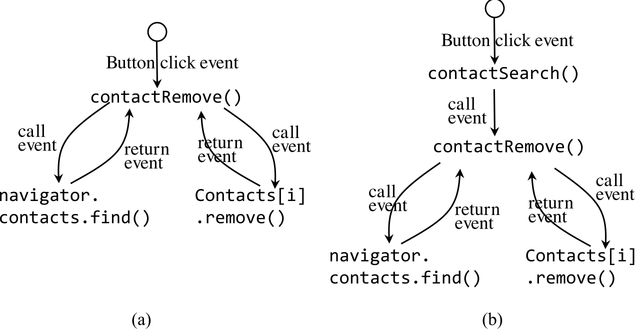


FIGURE 3. **Comparison of normal behaviors and malicious ones. a) Normal behavior of GroupMessageSender. b) Behavior of GroupMessageSender with code injected.**

The above example shows that function-activation information (e.g., call graph and triggering event) helps distinguishing anomalous behaviors from benign ones in hybrid apps.

IV. DESIGN

In this section, we describe the design of our approach. We first introduce the overall architecture of our solution, and the technique to extract function-activation information and system events. Finally, we present our key algorithms.

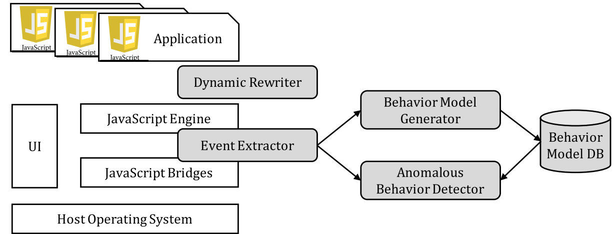


FIGURE 4. **Architecture of our approach.**

A. ARCHITECTURE

The architecture of our system is illustrated in Figure 4. It consists of the following modules: *Dynamic Rewriter*, *Event Extractor*, *Behavior Model Generator*, *Behavior Model Database*, and *Anomalous Behavior Detector*.

The dynamic rewriter is a run-time module inside the JavaScript engine. To get the function-activation information, such as function name, file path and function's line number, it intercepts the JavaScript code before the code is received by the JS engine, instruments the code, and passes the instrumented code to the JS engine. During the execution of the app, the instrumented code intercepts important function-activation information, which will then be used to build behavior model for the app. This is achieved through interaction with the event extractor, which is a run-time module of the JS engine. It extracts system interactions made by a Hybrid app. For example, when the app makes a PhoneGap API call or its button is clicked, these events will be intercepted and reported for building the behavior model.

Using events and function-level information extracted by the previous two components, the behavior model generator builds the behavior model of the app. Security analysts can run the app in a normal environment to obtain its original behavior model. After traversing the app and triggering as many behaviors as possible using a test suite, the output from the model generator will be treated as the app's original behavior model and stored in the behavior model database.

The anomalous behavior detector module matches the app behaviors against its original behavior model stored in the database. If the app's behaviors do not fit into the original model, it will be treated as anomalous.

B. FUNCTION BEHAVIOR MODEL

We define a function behavior model for hybrid mobile apps.

Definition (Function Behavior Model): The model proposed for hybrid apps is based on a state machine $(S, \Sigma, s_0, \delta, F)$*, where:*

- S *is a finite, non-empty set of states. For any state* $s = (s.id, s.attr) \in S$*,* $s.id$ *represents the state identifier;* $s.attr$ *represents the state attribute.*
- Σ *is a set of inputs. An input* $\sigma \in \Sigma$ *describes an event that triggers a state transition, where* $\phi \in \Sigma$ *means an empty input.*
- $s_0 \in S$ *is the initial state.*
- $\delta : S \times \Sigma \rightarrow S$ *is the state transition function. For two states* $s_a, s_b \in S$ *and* $\sigma \in \Sigma$*,* $s_b = \delta(s_a, \sigma)$ *represents the state transition from* s_a *to* s_b *triggered by the input* σ*.*
- F *is the set of final states, and* $F \subset S$*.*

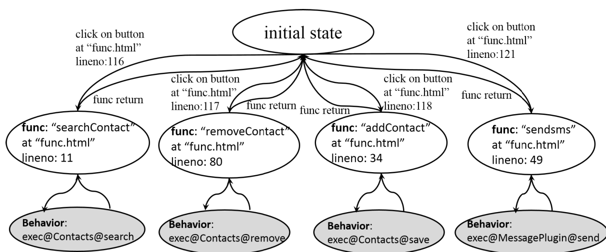


FIGURE 5. **The function behavior model for GroupMessageSender.**

Figure 5 is an example model of the motivating example. The state set of the app is represented as $S = \{$`initial state`, `searchContact`, `removeContact`, `addContact`, `sendSMS`, `exec@Contacts@search`, `exec@Contacts@remove`, `exec@Contacts@save`, `exec@MessagePlugin@send`$\}$ (For brevity, only $s.id$s are listed, and $s.attr$s are elaborated soon.).

There are two kinds of states in S: `searchContact`, `removeContact`, `addContact` and `sendSMS` are corresponding to internal JavaScript functions and `exec@Contact@search`, `exec@Contact@remove`, `exec@Contact@add` and `exec@MessagePlugin@send` are corresponding to calls to APIs, which are the system-level behaviors. As shown in Figure 5, $s.id$ of the function state `addContact` is the function name "addContact", $s.attr$ of `addContact` is `(func.html, 34)`, which means "addContact" is defined in the file "func.html" at the line number 34. Similarly, a line number (lineno X) hereafter refers to the line X in the file "func.html".

The set of trigger events is represented as $\Sigma = \{$"click on button at func.html lineno:116", "click on button at func.html lineno:117", "click on button at func.html lineno:118", "click on button at func.html lineno:121", "func return", $\phi\}$. "s_0" is "initial state" that we set for every app as the start point in state transition.

The arrowed line between two states represents a state transition. The trigger event of this transition is marked beside the arrowed line. As shown in Figure 5, for example, the transition from `initial state` to state `searchContact` is triggered by the event of "click on button at func.html lineno:116". According to the above definition, this transition can be recorded as: `searchContact` = δ(`initial state`, "click on button at func.html lineno:116"). As another example, to achieve the normal behavior "remove contact" (labeled as `exec@Contacts@remove`), this app has one state transition from `initial state` to `removeContact`, which is triggered by an event "click on button at func.html lineno:117".

In contrast, the injected malicious scripts directly call the function to trigger the "remove contact" behavior. This behavior can be detected by the behavior model. In the model, behavior "remove contact"(`exec@Contacts@remove`) is achieved by the state transition `initial state` $\rightarrow$ `searchContact` $\rightarrow$ `removeContact` with two trigger events "click on button at func.html lineno:116" and "HTMLImageElement.onerror at func.html lineno:2". The state transition in this attack is invalid. This is how we can use function activation information to effectively detect the code injection attacks in JavaScript applications.

C. EXTRACTION OF FUNCTION-ACTIVATION INFORMATION AND SYSTEM EVENTS

To completely model hybrid apps' behaviors, we need to intercept both function activation within the JavaScript engine, and the interaction between the JavaScript engine and its external environment.

To extract function-activation information, we instrument the JavaScript code before it is processed by the JavaScript engine. The instrumented JavaScript code needs to maintain

a virtual stack and reports the caller-callee relationship for building behavior models. Specifically, at the beginning of each function, the instrumented code will report the event of *entering* a function; at the end of each function, the instrumented code will report the event of *exiting* of the function. We will elaborate the implementation choice in Section V.

To extract system events, we monitor the activation of interface APIs in the bridge component. For example, calls to PhoneGap APIs, network requests, or UI events, will be reported by the event extractor. The reported function activation events and system information will be used in building behavior models and detecting malicious behaviors.

D. BEHAVIOR MODEL GENERATION

The *Behavior Model Generation* algorithm creates the behavior model from events extracted from the JavaScript environment. It takes as input a list of events (`Event-list`). It first creates an initial state s_0, where the identifier of s_0, $s_0.id = InitialState$, and the attribute of s_0, $s_0.attr = (null, null)$. Then the algorithm traverses the `Event-list`. If it finds a newly invoking event $e \in$ `Event-list` (in our implementation described in V, such events are marked as "func into"), it creates a new state s_i, where $s_i.id = \langle functionname \rangle$, and $s_i.attr = (\langle hostfile \rangle, \langle linenumber \rangle)$. It also creates a state transition function $s_i = \delta(s_{i-1}, \sigma_i)$, where $\sigma_i = \langle trigger\ event \rangle$. If the algorithm gets a return event $e \in$ `Event-list` (marked as "func out" in our implementation), then it creates a state transition function $s_i = \delta(s_{i-1}, \sigma_i)$, where $\sigma_i = function\ return$. If the algorithm finds a behavior event $e \in$ `Event-list` (marked as "behavior"), it creates a new state s_i, where $s_i.id = \langle behavior\ information \rangle$, and $s_i.attr = (null, null)$. It also creates a state transition function $s_i = \delta(s_i, \sigma_i)$, where $\sigma_i = null$. If the event belongs to none of above categories, the algorithm ignores this event and moves the next event in the `Event-list`. After the algorithm is done with the traverse of `Even-list`, a complete state-machine based behavior model of the hybrid application will be generated.

The behavior-model-creation algorithm we use in our approach is summarized in Algorithm 1.

Algorithm 1 Behavior Model Generation

Data: A list of events: `Event-list`
Result: Behavior model
1 **create** initial state s_0, where $s_0.id = InitialState$, and $s_0.attr = (null, null)$.
2 **while** *not at end of* `Event-list` **do**
3   **get** new event $e \in$`Event-list`;
4   **if** *event is marked as "func into"* **then**
5   **create** state s_i, where $s_i.id = \langle functionname \rangle$, and $s_i.attr = (\langle hostfile \rangle, \langle linenumber \rangle)$;
6   **create** state transition function $s_i = \delta(s_{i-1}, \sigma_i)$, where $\sigma_i = \langle trigger\ event \rangle$;
7   **else**
8   **if** *event is marked as "func out"* **then**
9   **create** state transition function $s_i = \delta(s_{i-1}, \sigma_i)$, where $\sigma_i = function\ return$;
10   **else**
11   **if** *event is marked as "behavior"* **then**
12   **create** new state s_i, where $s_i.id = \langle behavior\ information \rangle$, and $s_i.attr = (null, null)$;
13   **create** state transition function $s_i = \delta(s_i, \sigma_i)$, where $\sigma_i = null$;
14   **else**
15   **ignore** event;

E. ANOMALOUS BEHAVIOR DETECTION

After generating the behavior model of an app, we use an *Anomalous Behavior Detection* algorithm to detect whether its run-time behavior complies with the behavior model.

The detection algorithm we use in our approach is summarized in Algorithm 2. Our algorithm takes as inputs the behavior model of an app and the sequence of events to be checked. Given the original behavior model $M = (S, \Sigma, \delta, s_0, F)$ and the behavior sequence to be checked Σ', the algorithm traverses the state of M, driven by events in Σ' (line 4-12). If an event σ'_i in Σ' drives the current state to a state which is valid according to the transition in M (line 5), the transition is regarded legitimate. On contrary, if an event σ'_i in Σ' does not lead M to a correct state, meaning that a state beyond the transition allowed by M is reached (which may be because of injected functions), our algorithm outputs the captured abnormal state (line 10-11). Note that at a particular state, there may be multiple target states. We thus use a *Stack* (line 2, 6 and 9) to maintain this state entering/exiting information.

Algorithm 2 Anomalous Behavior Detection

Data: Original behavior model and test behavior
Result: Abnormal behavior detection result
1 **given** original behavior model M and testing behavior Σ', **let** the number of states in M, $||S|| = n$ and the number of events $||\Sigma'|| = e'$.
2 $Stack.init()$
3 **let** the current state $s \leftarrow s_0$
4 **foreach** $\sigma'_i \in \Sigma'$ *in temporal sequence*, $0 \leq i \leq e'$ **do**
5   **if** *(σ'_i is a function enter event or σ'_i is a UI event) and $\exists$ state $s_t \in S$ s.t. $s_t = \delta(s, \sigma'_i)$* **then**
6     $Stack.push(s)$
7     $s \leftarrow s_t$
8   **else if** *σ'_i is a function exit event* **then**
9     $s \leftarrow Stack.pop()$
10   **else**
11   **report** abnormal state.
12   $i \leftarrow i + 1$

V. IMPLEMENTATION

We have implemented our solution targeting hybrid Android applications which use PhoneGap as the bridge.

A. INTERCEPTING FUNCTION ACTIVATION

We leverage the mechanism of the exception handling in V8 to extract function-activation information. For every exception, V8 records the function information, i.e., function name, file path and line number, where the exception is thrown. We intercept and rewrite the JavaScript code of the apps, such that it throws an exception in every function. In this way, we get the function call stack during the execution of the rewritten code. The call stack will then be used as the context of apps' behaviors. We used two third-party software to assist rewriting JavaScript code, JXcore [55] and AST-query [56]. JXcore is a JavaScript runtime environment which enables AST-query to run. AST-query reads JavaScript code and rewrites it according to our instructions.

Specifically, for every JavaScript function of a given app, the dynamic rewriter inserts code to get the JavaScript call stack at the beginning of the function and log the entering event of the function, and log the exit event at the end of the function. The call stack contains function name (which is used as the state identifier), file path and line number (which is used as the state attribute). It also contains the function's caller (which is the trigger of state transition).

In the source code of Android, the function *evaluate()* (which is in the source file *WebCore/bindings/v8/V8Proxy.cpp*) is where WebView starts to execute JavaScript code. We modify this function to intercept and rewrite JavaScript code before passing the instrumented code to V8.

B. EXTRACTING SYSTEM EVENTS

The event extractor monitors and records the APIs invoked by the app. In the WebKit component of the Android system, we instrument hooks to extract the function information. More specifically, for the app's *local behaviors* triggered by calling certain PhoneGap APIs, we modify the function `npObjectInvokeImpl()` in the source file *WebCore/binding/v8/V8NPObject.cpp* to extract the API calls made by PhoneGap, as well as the corresponding system resources that the app requests. For the app's *network behaviors*, we modify the function `createRequest()` in the source file *WebCore/xml/XMLHttpRequest.cpp* where the XML HTTP requests are generated, to extract the information about Ajax requests.

VI. EVALUATION

In this section, we evaluate the effectiveness and performance of our solution.

A. EFFECTIVENESS

In order to evaluate the effectiveness of our solution, we deploy our approach to model the behaviors of real-world popular hybrid Android apps, and demonstrate its capabilities in anomalous behavior detection with simulated code injection attacks on those apps. We present two case studies showing how injected behaviors are detected.

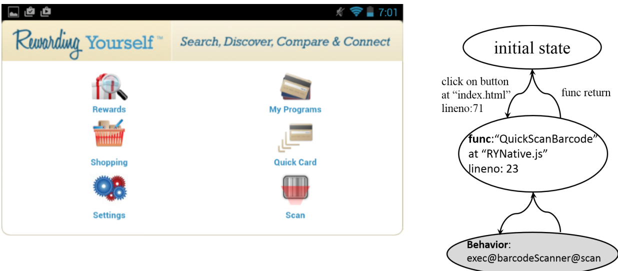


FIGURE 6. The RewardingYourself application and its original behavior model.

1) CASE STUDY I: REWARDINGYOURSELF

RewardingYourself [57] is a hybrid app that tracks millage points of the loyalty program of users. The original behavior model for RewardingYourself is shown in Figure 6, where nodes are states represented by related information of JavaScript functions in the app, and the arrow from a state to another denotes the transition between these two states. Our system also marks the arrow with trigger which causes the state transition.

```
<img src="x" onerror ="
  var options = {maximumAge:30000, timeout:100000,
      enableHighAccuracy:false};
  navigator.geolocation.watchPosition( //Get user's
      current position
    function onSuccess(position) {
      var pos = 'longitude:' + position.coords.
          longitude + ';latitude:' +
               position.coords.latitude;
      xmlhttp = new XMLHttpRequest(); //Send the
          position to remote server
      xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState==4 && xmlhttp.status
            ==200) {
          alert('send message: ' + pos);
        }
      };
      xmlhttp.open('GET', 'http://192.168.0.106/info
          .php?pos=' + pos, true);
      xmlhttp.send();
    }, function () {}, options
  );
">
```

FIGURE 7. A snippet of malicious code which pops up location information.

We use a malicious QR code, which embeds the malicious code shown in Figure 7. The attack will cause an alert box to be popped out with the location information, shown in Figure 8. Under the context `initial state`, the attack causes new states such as `exec@geolocation@getCurrentPosition` and `exec@HTTPRequest@Get@http://192.168.0.106/info.php?msg=Longitude, 116.372048.Latitude, 39.892778.@true` that deviates from the behavior model. Our approach detects it and raises an alarm to prevent the behavior from being executed.

TABLE 1. Performance evaluation (in seconds).

Scenario	No.	Task 1 Time Web Page Load	Task 2 Time PhoneGap API invoke	Total Time
Scenario 1 Pristine Android System	1	1.08	4.31	5.39
	2	1.09	4.42	5.51
	3	1.12	4.32	5.44
	4	1.20	4.63	5.83
	5	0.91	4.62	5.53
	avg	1.08(100%)	4.46(100%)	5.54(100%)
Scenario 2 With Our Approach	1	5.27	4.11	9.38
	2	5.37	4.03	9.40
	3	4.78	5.70	10.48
	4	5.24	4.58	9.82
	5	5.26	4.15	9.41
	avg	5.18(480%)	4.51(101.2%)	9.70(175.1%)

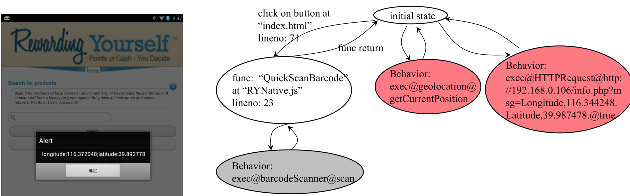
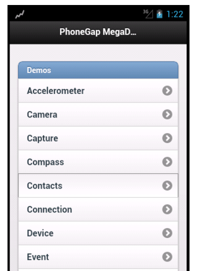


FIGURE 8. The result of code injection attack to RewardingYourself and the resulting functional call behaviors.

2) CASE STUDY II: PHONEGAPMEGA

PhoneGapMega [58] demonstrates features of PhoneGap APIs, which contains examples of using almost all APIs of PhoneGap. A fragment of the behavior model of PhoneGap-Mega is shown in Figure 9.



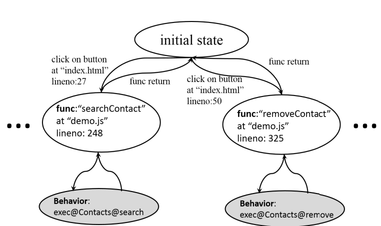


FIGURE 9. The PhoneGapMega application and a fragment of the original behavior model of PhoneGapMega.

PhoneGapMega has the same type of vulnerabilities that allows malicious code injection. Malicious JavaScript (shown in Figure 10) injected into a contact's name can be executed in the app. When the app reads the malicious contact item, the injected code will be executed. The result of the execution will read the whole contact list of the victim user and send them to a specific remote server. Shown in Figure 11, the attack has new states that do not exist in the normal model, which triggers the alert of our detection.

B. PERFORMANCE EVALUATION

We build a test app that has the functionality of manipulating contacts and sending SMS. Instead of manually clicking on buttons to trigger these functionalities, this test app automatically finishes the following two tasks:

1) loading app's web page, while the dynamic rewriter rewrites the JavaScript code
2) adding a new contact, removing this newly added contact, searching the contact list, and sending SMS to the first contact member in the list, while the event extractor extracts the information about invoking corresponding PhoneGap APIs.

We run this app in two scenarios: one with pristine Android system, another with our approach, and measure the time it takes from when the app starts to when the above tasks finishes. We repeat task 2) ten times to increase the time difference between these two scenarios, and make the time measurement more accurate.

```
<img src="x" onerror="
  var fields = ['*'];
  var options = new ContactFindOptions();
  options.multiple = true;
  navigator.contacts.find(fields, //Read user's
      contact list
    function(contacts) {
      var msg = '';
      for (var i = 0; i <= contacts.length - 1; i
          ++) {
        msg += contacts[i].displayName + ', ' +
            contacts[i].phoneNumbers[0].value +
               ', ' + contacts[i].emails[0].value +
                     '.';
      }
      xmlhttp = new XMLHttpRequest(); //Send the
          contact list to remote server
      xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.
            status == 200) {
          alert(xmlhttp.responseText);
        }
      }
      xmlhttp.open('GET', 'http://192.168.0.106/info
          .php?msg=' + msg, true);
      xmlhttp.send();
    }, function() {}, options
  );
"/>
```

FIGURE 10. A snippet of malicious code which steals contacts.

Table 1 illustrates the evaluation result, in which the web page loading time is increased. However, since web pages

FIGURE 11. The result of code injection attack to PhoneGapMega and the corresponding function call behaviors.

are often loaded once in the beginning of running an app, this amount of time increase is acceptable. In addition, the time taken to invoke the PhoneGap APIs is increased slightly, which might not be noticed by the users at all. Above all, the performance overhead introduced by our approach is reasonable.

C. DISCUSSION OF LIMITATION

The detection in our approach is based on function call relationship and triggering events. Despite the effectiveness shown by the case studies, it is possible for attackers to inject the code and active it under the same condition as in the original app. In such cases, our approach may miss the injection within a target function. We take as future work to design finer-grained behavior model to capture those low-level injection behaviors.

VII. CONCLUSION

Using JavaScript-based technologies to build mobile apps is a popular technique in the web and social network infrastructure. However, the flexibility of the JavaScript language introduces new security challenges in these platforms. In this paper, we propose to detect malicious JavaScript behaviors in JavaScript applications. The behavior model captures an application's behaviors as well as their function level execution information. Our prototype detection system can automatically build the behavior models for hybrid apps to detect anomalous behaviors. We demonstrate its effectiveness of anomalous behaviors detection with case studies on real-world hybrid apps. As a future work, we will investigate how our solutions can be adopted by JavaScript applications in other domains, such as server-side applications and IoT solutions.

REFERENCES

[1] Z. Cai, Z. He, X. Guan, and Y. Li, "Collective data-sanitization for preventing sensitive information inference attacks in social networks," *IEEE Trans. Depend. Sec. Comput.*, to be published.

[2] (Dec. 2015). *Cross-Site Scripting (XSS)*. Accessed: Jan. 2018. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[3] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2014, pp. 66–77.

[4] A. Castiglione, R. De Prisco, and A. De Santis, "Do you trust your phone?" in *E-Commerce and Web Technologies*. Berlin, Germany: Springer, 2009.

[5] A. Armando, A. Merlo, and L. Verderame, "An empirical evaluation of the Android security framework," in *Security and Privacy Protection in Information Processing Systems*. Berlin, Germany: Springer, 2013, pp. 176–189.

[6] B. V. Livshits and M. S. Lam, "Finding security errors in java program with static analysis," in *Proc. 14th Usenix Secur. Symp.*, Baltimore, MD, USA, 2005, pp. 1–16.

[7] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proc. USENIX Secur.*, vol. 6. 2006, pp. 179–192.

[8] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting Web application vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, 2006, p. 6.

[9] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Germany: Springer, 2008, pp. 23–43.

[10] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening Web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing*. Berlin, Germany: Springer, 2005.

[11] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection*. Berlin, Germany: Springer, 2006, pp. 124–145.

[12] R. Sekar, "An efficient black-box technique for defeating Web application attacks," in *Proc. NDSS*, 2009, pp. 1–17.

[13] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-side detection of cross-site scripting attacks," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2008, pp. 335–344.

[14] (Dec. 2015). *Noscript Features: Anti-XSS Protection*. Accessed: Jan. 2018. [Online]. Available: http://noscript.net/features#xss

[15] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks," in *Proc. NDSS*, 2009, pp. 1–4.

[16] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 66–77.

[17] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, May 2001, pp. 144–155.

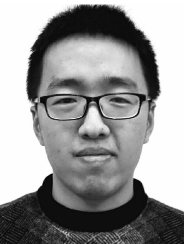
[18] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Washington, DC, USA, 2003, pp. 62–75. [Online]. Available: http://dl.acm.org/citation.cfm?id=829515.830554

[19] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for Ajax intrusion detection," in *Proc. 18th Int. Conf. World Wide Web (WWW)*, New York, NY, USA, 2009, pp. 561–570. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526785

[20] X. Dong, K. Patil, J. Mao, and Z. Liang, "A comprehensive client-side behavior model for diagnosing attacks in Ajax applications," in *Proc. 18th Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, Jul. 2013, pp. 177–187.

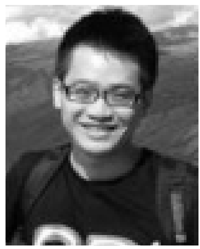
[21] (Jan. 2015). *PhoneGap Official Site*. Accessed: Jan. 2018. [Online]. Available: http://phonegap.com/
[22] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks," in *Proc. NDSS Symp.*, 2014, p. 1.
[23] X. Jin, L. Wang, T. Luo, and W. Du, "Fine-grained access control for HTML5-based mobile applications in android," in *Proc. 16th Inf. Secur. Conf. (ISC)*, 2013, pp. 309–318.
[24] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. IEEE Symp. Secur. Privacy*, May 1996, pp. 120–128.
[25] J. Mao, R. Wang, Y. Chen, and Y. Jia, "Detecting injected behaviors in HTML5-based Android applications," *J. High Speed Netw.*, vol. 22, no. 1, pp. 15–34, 2016.
[26] J. Mao, R. Wang, Y. Chen, Y. Xiao, Y. Jia, and Z. Liang, "A function-level behavior model for anomalous behavior detection in hybrid mobile applications," in *Proc. Int. Conf. Identificat., Inf. Knowl. Internet Things*, 2016, pp. 1–9.
[27] (Jan. 2015). *Wiki on the Same-Origin Policy*. Accessed: Jan. 2018. [Online]. Available: https://en.wikipedia.org/wiki/Same-origin_policy
[28] (Dec. 2015). *Introducing Content Security Policy*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy
[29] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng. (ICSE)*, May 2008, pp. 171–180.
[30] D. Balzarotti *et al.*, "Saner: Composing static and dynamic analysis to validate sanitization in Web applications," in *Proc. IEEE Symp. Security Privacy (SP)*, May 2008, pp. 387–401.
[31] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proc. ACM Symp. Appl. Comput.*, 2006, pp. 330–337.
[32] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proc. NDSS*, 2007, p. 12.
[33] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense," in *Proc. NDSS*, 2009, p. 20.
[34] M. T. Louw and V. N. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 331–346.
[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Trans. Web*, vol. 1, no. 3, p. 11, 2007.
[36] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 237–249, 2007.
[37] (Dec. 2015). *Microsoft Web Sandbox*. [Online]. Available: http://websandbox.livelabs.com/
[38] (Dec. 2015). *Google Caja*. Accessed: Jan. 2018. [Online]. Available: https://code.google.com/p/google-caja/
[39] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1193–1204.
[40] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 655–670.
[41] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proc. Found. Softw. Eng. (FSE)*, 2015, pp. 272–283.
[42] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "DexterJS: Robust testing platform for DOM-based XSS vulnerabilities," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 946–949.
[43] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang, "You can't be me: Enabling trusted paths and user sub-origins in Web browsers," in *Research in Attacks, Intrusions and Defenses*. Berlin, Germany: Springer, 2014, pp. 150–171.
[44] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang, "Protecting sensitive Web content from client-side vulnerabilities with CRYPTONS," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1311–1324.
[45] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song, "Data-confined HTML5 applications," in *Computer Security—ESORICS*. Berlin, Germany: Springer, 2013, pp. 736–754.
[46] X. Dong, M. Tran, Z. Liang, and X. Jiang, "AdSentry: Comprehensive and flexible confinement of JavaScript-based advertisements," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 297–306.
[47] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen, "PathCutter: Severing the self-propagation path of XSS javascript worms in social Web networks," in *Proc. NDSS*, 2012, pp. 1–14.
[48] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in HTML5 applications," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, p. 23.
[49] L. Backstrom, C. Dwork, and J. Kleinberg, "Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 181–190.
[50] E. Zheleva and L. Getoor, "To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles," in *Proc. 18th Int. Conf. World Wide Web*, 2009, pp. 531–540.
[51] M. Balduzzi, C. Platzer, T. Holz, E. Kirda, D. Balzarotti, and C. Kruegel, "Abusing social networks for automated user profiling," in *Recent Advances in Intrusion Detection*. Berlin, Germany: Springer, 2010, pp. 422–441.
[52] A. Chaabane *et al.*, "You are what you like! Information leakage through users' interests," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2012, pp. 1–14.
[53] (Apr. 2015). *RhoMobile Official Site*. Accessed: Jan. 2018. [Online]. Available: http://rhomobile.com/
[54] (Feb. 2015). *Appcelerator Official Site*. Accessed: Jan. 2018. [Online]. Available: http://www.appcelerator.com/
[55] (Dec. 2015). *JXcore*. Accessed: Jan. 2018. [Online]. Available: https://github.com/jxcore/jxcore
[56] (Dec. 2015). *Ast-Query*. Accessed: Jan. 2018. [Online]. Available: https://github.com/SBoudrias/ast-query
[57] (May 2014). *Rewarding Yourself*. [Online]. Available: https://apkpure.com/rewardingyourself/com.loyaltymatch.rewardingyourself
[58] (Jun. 2014). *Phonegap Mega*. Accessed: Jan. 2018. [Online]. Available: https://play.google.com/store/apps/details?id=com.camden.phonegapmega

JIAN MAO received the B.S. and Ph.D. degrees from Xidian University, China. She is currently an Assistant Professor with the School of Electronic and Information Engineering, Beihang University, Beijing, China. Her research interests include cloud security, Web security, and mobile security.



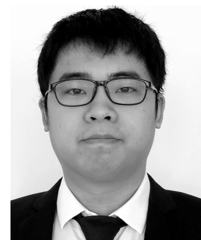
JINGDONG BIAN received the B.S. degree in electronic and information engineering from Jilin University, Jilin, China, in 2016. He is currently pursuing the master's degree in electronic and information engineering with Beihang University, Beijing, China. His research interests include Web security, mobile security, and privacy analysis.

GUANGDONG BAI received the bachelor's and master's degrees in computing science from Peking University, China, in 2008 and 2011, respectively, and the Ph.D. degree in computing science from the National University of Singapore (NUS) in 2015. He was a Post-Doctoral Research Fellow with NUS. He has been a Faculty Member with the Singapore Institute of Technology since 2016. His research interests include mobile security, protocol verification, and formal methods on security.



RUILONG WANG received the B.S. and M.S. degrees in electronic and information engineering from Beihang University, Beijing, China, in 2013 and 2016, respectively. His research interests include software security, Web security, mobile security, and program analysis.

YUE CHEN received the B.S. and M.S. degrees in electronic and information engineering from Beihang University, Beijing, China, in 2012 and 2016, respectively. His research interests include Web security and mobile security.



YINHAO XIAO received the bachelor's degree in information and computing science from the Guangdong University of Technology in 2012, and the master's degrees in applied mathematics and in computer science from The George Washington University in 2014 and 2015, respectively, where he is currently pursuing the Ph.D. degree with the Department of Computer Science. His research interests include system security and social network privacy.

ZHENKAI LIANG received the B.S. degree from Peking University in 1999 and the Ph.D. degree from Stony Brook University in 2006. He is currently an Associate Professor with the Department of Computer Science, National University of Singapore. His research interests include software security, Web security, and mobile security.