2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Towards Understanding and Reasoning about Android Interoperations

Sora Bae
KAIST
Daejeon, South Korea
bsora92@kaist.ac.kr

Sungho Lee
KAIST
Daejeon, South Korea
eshajlee@gmail.com

Sukyoung Ryu
KAIST
Daejeon, South Korea
sukyoung.ryu@gmail.com

***Abstract*—Hybrid applications (apps) have become one of the most attractive options for mobile app developers thanks to its support for portability and device-specific features. Android hybrid apps, for example, support portability via JavaScript, device-specific features via Android Java, and *seamless interactions* between them. However, their interoperation semantics is often under-documented and unintuitive, which makes hybrid apps vulnerable to errors. While recent research has addressed such vulnerabilities, none of them are based on any formal grounds.**

In this paper, we present the first formal specification of Android interoperability to establish a firm ground for understanding and reasoning about the interoperations. We identify its semantics via extensive testing and thorough inspection of Android source code. We extend an existing multi-language semantics to formally express the key features of hybrid mechanisms, *dynamic* and *indistinguishable* interoperability. Based on the extensions, we incrementally define a formal interoperation semantics and disclose its numerous unintuitive and inconsistent behaviors. Moreover, on top of the formal semantics, we devise a lightweight type system that can detect bugs due to the unintuitive inter-language communication. We show that it detects more bugs more efficiently than HybriDroid, the state-of-the-art analyzer of Android hybrid apps, in real-world Android hybrid apps.

***Index Terms*—Android hybrid applications, interoperability, multi-language systems, operational semantics, type system**

I. Introduction

Taking the best out of both web and native apps—portability and device-specific features—*hybrid apps* have become prevalent [1], [2]. In hybrid apps, JavaScript code handles user interactions to be run on multiple browsers, native code accesses device-specific features, such as accelerometer, and they can communicate via interoperations.

However, understanding the interoperation behaviors of hybrid apps is not trivial because the implementation languages have different semantics. Because the semantics of JavaScript is different from that of native languages like Android Java, it is quite easy for developers to make errors like passing wrong number or wrong types of arguments to foreign function calls. Consider Fig. 1 an excerpt from the Adobe Acrobat Reader source code, which is downloaded using the PlayDrone project [3]. It shows an interaction between Java[1] and JavaScript. Fig. 1(a) shows Java code where class `ARJSApp` declares method `alert` that can be used in JavaScript

```
class ARJSApp {
  @JavascriptInterface
  public final int alert(String paramString,
                         int paramInt) { ... }
  ...
}
```
(a) Java class `ARJSApp` declares method `alert` that can be injected to JavaScript contexts.

```
this.mWebView.addJavascriptInterface(
  new ARJSApp(this), "_app");
```
(b) Java code injects a Java object `new ARJSApp(this)` into the JavaScript context with name `_app`.

```
App.prototype.alert = function(cMsg, nIcon) {
    return window._app.alert(cMsg + "", nIcon);
};
var app = new App();
app.alert(cAlert);
```
(c) JavaScript invokes the injected Java method `window._app.alert`.

Fig. 1: Interaction between Android Java and JavaScript

as denoted by annotation `@JavascriptInterface`. Fig. 1(b) shows another Java code where an instance of `ARJSApp` is injected to the JavaScript context as top-level variable `_app` via method `addJavascriptInterface` of class `WebView`. Then, Fig. 1(c) illustrates an actual interaction in JavaScript code. JavaScript object `app` calls its function property `alert` with a single argument `cAlert` while the function declaration expects two parameters `cMsg` and `nIcon`. Calling a function expecting two arguments with only one argument is a valid semantics in JavaScript: the second parameter `nIcon` gets value `undefined`. Then, the function body calls the injected Java method `window._app.alert` expecting a string and an integer with `cMsg + ""` and `nIcon`. The second argument from JavaScript does not satisfy the type requirement of the Java method. Moreover, inter-language communication introduces various design choices to make. Consider Fig. 1 once again. When value `undefined` is passed to function `window._app.alert`, would it throw an exception? If method `alert` is overloaded in the Java side, would it be still overloaded after the injection to the JavaScript side, which does not support overloading? When the same Java object is injected to JavaScript twice as `_app` and `app_`, would the injected objects be still the same object in the JavaScript side? And so on.

However, existing research for Android hybrid apps [4]–

[1]While Android Java is different from Java, we use Android Java and Java interchangeably when their differences do not matter.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00038

[7] and even the official Android documentation [8] do not clearly specify the interoperation behaviors and its design choices. The existing works focus on handling real-world Android apps and various engineering decisions for scalability; they do not aim to cover the inter-communication semantics. The state-of-the-art static analyzer for Android hybrid apps, HybriDroid [7], analyzes inter-communication between Android Java and JavaScript, and detects misuses of foreign function calls, but it addresses interoperations partially; it does not consider, for example, multiple injections of single Java object. The Android documentation is neither exhaustive nor consistent; it does not address any design choices discussed above.

In this paper, we present the first formal specification of Android interoperability to provide a firm ground for understanding and reasoning about inter-language communication in Android hybrid apps. We identify the interoperation semantics via extensive testing and inspection of the publicly available Android source code [9]. To formally specify its key features, *dynamic* and *indistinguishable* interoperability, we extend the existing multi-language calculus [10]. For dynamic interoperations, the extension dynamically builds *bridge*s across Java and JavaScript, which allow accesses to Java objects from JavaScript. For indistinguishable interoperations, it creates a JavaScript object for a bridge to support an interchangeable switch of languages from JavaScript to Java. Based on the extension, we present an operational semantics of Android interoperability. Moreover, we discuss its numerous *unintuitive and inconsistent* behaviors that have not been disclosed by the existing research.

Taking advantage of the semantics, we also present a light-weight type system that detects *interoperation bugs* including MethodNotFound bugs defined by HybriDroid. While HybriDroid uses an expensive points-to analysis, our type system is light-weight with linear time traversal over target apps. We evaluate our type system on real-world Android hybrid apps, and show that it detects about twice as many true bugs than HybriDroid with the average speed-up of 20.7x compared to HybriDroid.

The contributions of this paper include the following:

- **We identify complex but under-documented behaviors in Android hybrid apps.** It requires extensive testing and thorough investigation of Android source code due to incomplete Android documentation.
- **We extend the existing multi-language semantics for dynamic and indistinguishable interoperability.** It may give an example to specify various inter-language communications beyond the Android hybrid mechanism.
- **We present the first formal semantics for Android interoperability.** It provides a formal ground to reason about Android hybrid apps, and it allows us to uncover their unintuitive, surprising, and inconsistent behaviors.
- **We develop a precise and efficient type system that detects interoperation bugs.** The type system detects more true bugs more efficiently than HybriDroid.

In the rest of the paper, we present key interoperation features of hybrid apps (Section II) and a formal operational semantics of Android interoperability from core to wild behaviors (Section III). Using the formal semantics, we present a type system for safe interoperability in Android hybrid apps (Section IV). We evaluate the type system (Section V), discuss related work (Section VI), and conclude (Section VII).

II. Android Interoperability

We describe the Android hybrid mechanism [8] and introduce its key features that are not expressible in the existing multi-language semantics [10].

A. From Android Java to JavaScript

Android Java code can execute JavaScript code using two methods. First, `evaluateJavascript` takes two arguments:

```
public void evaluateJavascript(
  String script, ValueCallback<String> resultCallback)
```

It executes the first string argument denoting JavaScript code, and invokes the second callback, if any, when the execution completes. Second, `loadUrl` loads a given URL:

```
public void loadUrl(String url)
```

possibly with given HTTP request headers for the URL:

```
public void loadUrl(
  String url, Map<String,String> additionalHttpHeaders)
```

It executes JavaScript code embedded in the web page at the URL. To focus on the core interaction functionality, we ignore the callback argument for `evaluateJavascript` and the HTTP request headers argument for `loadUrl`. Also, while `loadUrl` can take a string argument representing JavaScript code rather than a URL, we ignore such case because it is the same as using `evaluateJavascript`. We assume that Java and JavaScript code run synchronously for simplicity.

B. From JavaScript to Android Java

JavaScript code executed by `evaluateJavascript` or `loadUrl` can invoke Java methods via objects "injected" by method `addJavascriptInterface`:

```
public void addJavascriptInterface(Object object, String name)
```

It injects the first object argument into the JavaScript context using the second string argument as a name, which then behaves as a *bridge* for inter-language communication. Injected Java objects can be removed from the JavaScript context by the following method:

```
public void removeJavascriptInterface(String name)
```

For example, an instance of class `Bridge` can be injected to the JavaScript context by `addJavascriptInterface(new Bridge(), "brg")`, and it is mapped as a JavaScript object named `brg`. We call such mapped JavaScript objects *bridge objects*.

JavaScript code invokes Java methods via bridge objects in the same way as it calls function properties of JavaScript objects. For example, for Java method `foo` in class `Bridge`, JavaScript can invoke the method by `brg.foo()`. While Android versions 4.2 and above allow only public methods annotated with `@JavascriptInterface` to be accessible from

JavaScript, we assume that every method is accessible from JavaScript for simplicity. Note that Java fields are not accessible from JavaScript.

C. Dynamic and Indistinguishable Interoperability

We identify two distinct key features in the Android hybrid mechanism: *dynamic* and *indistinguishable* interoperability, which are not expressible in existing multi-language semantics [10]. We call the existing semantics MF in this paper.

First, Java code accessible from JavaScript changes dynamically, for example, by `addJavascriptInterface` and `removeJavascriptInterface`. For example, access to class `Bridge` from JavaScript code varies over runtime. On the other hand, MF can statically determine the accessibility via fixed and static bridges. Note that Android supports flexible and dynamic bridges for inter-language communication.

Second, JavaScript code invokes Java methods in the same way as it calls function properties of JavaScript objects. Such syntatically indistinguishable interoperations switch languages interchangeably while MF introduces a new kind of expressions called "syntatic boundaries" to support a switch of languages.

III. Android Interoperability Semantics

We present Android interoperation semantics by extending MF with bridges. For simplicity, we omit Android-specific features like activity life cycles.

A. Models for Java and JavaScript

We first present base models for Java and JavaScript in Fig. 2. Following the convention of MF, we have typeset non-terminals of Java model using a **bold font with serifs**, and those of JavaScript model with a light sans-serif font. Occasionally, we use subscripts and superscripts when the font difference is too subtle to differentiate Java and JavaScript: *V* for Java and *S* for JavaScript. Java and JavaScript models are based on Featherweight Java [11] and λ_{JS} [12], respectively.

Java code $\mathbf{p}$ consists of a sequence of object declarations $\overline{\mathbf{od}}$ followed by a sequence of expressions $\overline{\mathbf{e}}$. For presentation brevity, we use singleton objects instead of classes because the interoperations do not utilize features like class inheritance, constructors, and fields. An object declaration contains a sequence of method declarations $\overline{\mathbf{md}}$. We use $\mathbf{A}$ for an object name and $\mathbf{m}$ for a method name. A method body is an expression $\mathbf{e}$, which is either a constant $\mathbf{c}$, a variable $\mathbf{x}$, or method invocation. A constant is a number $\mathbf{n}$, an object `[`$\mathbf{A}$`]`, or `null`. A Java type τ^v may be a primitive type ι, an object $\mathbf{A}$, or `void` for internal value `()`. We describe the operational semantics in the Felleisen-Hieb small-step operational semantics with evaluation contexts [13]. The operational semantics is conventional using the following:

$$\texttt{MatchType}(\mathbf{v},\tau^v) = \begin{cases} \texttt{true} & \text{if } (\mathbf{v} = \mathbf{n} \quad\wedge\ \tau^v = \iota) \\ & \vee\ (\mathbf{v} = \texttt{[}\mathbf{A}\texttt{]} \ \wedge\ \tau^v = \mathbf{A}) \\ & \vee\ (\mathbf{v} = \texttt{null} \wedge\ \tau^v = \mathbf{A} \text{ for any } \mathbf{A}) \\ & \vee\ (\mathbf{v} = \texttt{()} \quad\ \wedge\ \tau^v = \texttt{void}) \\ \texttt{false} & \text{otherwise} \end{cases}$$

that checks whether a given value has a given type.

$$\begin{array}{lcl}
\mathbf{p} & ::= & \overline{\mathbf{od}}\ \overline{\mathbf{e}} \\
\mathbf{od} & ::= & \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \\
\mathbf{md} & ::= & \tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \\
\mathbf{e} & ::= & \mathbf{c} \mid \mathbf{x} \mid \mathbf{e}\,.\,\mathbf{m}(\overline{\mathbf{e}}) \\
\mathbf{c} & ::= & \mathbf{n} \mid \texttt{[}\mathbf{A}\texttt{]} \mid \texttt{null} \\
\mathbf{v} & ::= & \mathbf{c} \mid \texttt{()} \\
\tau^v & ::= & \iota \mid \mathbf{A} \mid \texttt{void} \\
\mathbf{E} & ::= & []_V \mid \mathbf{E}\,.\,\mathbf{m}(\overline{\mathbf{e}}) \mid \mathbf{v}\,.\,\mathbf{m}(\overline{\mathbf{v}}\ \mathbf{E}\ \overline{\mathbf{e}})
\end{array}$$

$$[\text{Jv-Call}]\quad \frac{\begin{array}{cc}\texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} & \tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \in \{\ \overline{\mathbf{md}}\ \} \\ |\overline{\mathbf{x}}| = |\overline{\mathbf{v}}| & \overline{\texttt{MatchType}(\mathbf{v},\tau^v)}\end{array}}{\mathcal{E}[\texttt{[}\mathbf{A}\texttt{]}\,.\,\mathbf{m}(\overline{\mathbf{v}})]_V \to\ \mathcal{E}[\mathbf{e}[\overline{\mathbf{v}/\mathbf{x}}]]}$$

(a) Java model

$$\begin{array}{lcl}
\mathsf{e} & ::= & \mathsf{c} \mid \mathsf{x} \mid \texttt{let}\ \mathsf{x} = \mathsf{e}\ \texttt{in}\ \mathsf{e} \mid \mathsf{e}\,.\,\mathsf{m} \mid \mathsf{e}\,.\,\mathsf{m} = \mathsf{e} \mid \mathsf{e}\,.\,\mathsf{m}(\overline{\mathsf{e}}) \\
\mathsf{c} & ::= & \mathsf{n} \mid \texttt{[}\mathsf{A}\texttt{]} \mid \texttt{function}\ (\overline{\mathsf{x}})\ \{\mathsf{e}\} \mid \texttt{null} \\
\mathsf{v} & ::= & \mathsf{c} \mid \texttt{undefined} \\
\mathsf{E} & ::= & []_S \mid \texttt{let}\ \mathsf{x} = \mathsf{E}\ \texttt{in}\ \mathsf{e} \mid \mathsf{E}\,.\,\mathsf{m} \mid \mathsf{E}\,.\,\mathsf{m} = \mathsf{e} \mid \mathsf{v}\,.\,\mathsf{m} = \mathsf{E} \\
& \mid & \mathsf{E}\,.\,\mathsf{m}(\overline{\mathsf{e}}) \mid \mathsf{v}\,.\,\mathsf{m}(\overline{\mathsf{v}}\ \mathsf{E}\ \overline{\mathsf{e}})
\end{array}$$

$$[\text{Js-Let}]\quad \mathcal{O}\ \mathcal{E}[\texttt{let}\ \mathsf{x} = \mathsf{v}\ \texttt{in}\ \mathsf{e}]_S \to \mathcal{O}\ \mathcal{E}[\mathsf{e}[\mathsf{v}/\mathsf{x}]]$$

$$[\text{Js-Load1}]\quad \frac{\mathcal{O}(\mathsf{A}\,.\,\mathsf{m}) = \mathsf{v}}{\mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}\,.\,\mathsf{m}]_S \to \mathcal{O}\ \mathcal{E}[\mathsf{v}]}$$

$$[\text{Js-Load2}]\quad \frac{\mathsf{A}\,.\,\mathsf{m} \notin \mathit{dom}(\mathcal{O})}{\mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}\,.\,\mathsf{m}]_S \to \mathcal{O}\ \mathcal{E}[\texttt{undefined}]}$$

$$[\text{Js-Store}]\quad \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}\,.\,\mathsf{m} = \mathsf{v}]_S \to \mathcal{O}[\mathsf{A}\,.\,\mathsf{m} := \mathsf{v}]\ \mathcal{E}[\mathsf{v}]$$

$$[\text{Js-Call}]\quad \frac{\mathcal{O}(\mathsf{A}\,.\,\mathsf{m}) = \texttt{function}\ (\overline{\mathsf{x}})\ \{\mathsf{e}\}}{\mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}\,.\,\mathsf{m}(\overline{\mathsf{v}})]_S \to \mathcal{O}\ \mathcal{E}[\mathsf{e}[\overline{\mathsf{v}/\mathsf{x}}]]}$$

(b) JavaScript model

Fig. 2: Models for Java and JavaScript

$$\begin{array}{lcl}
\mathbf{p} & ::= & \overline{\mathbf{od}}\ \overline{\mathbf{t}} \\
\mathbf{t} & ::= & \mathbf{e} \mid \texttt{add(}\texttt{[}\mathbf{A}\texttt{]}\texttt{)} \mid \texttt{remove(}\texttt{[}\mathbf{A}\texttt{]}\texttt{)} \mid \texttt{evaluate(}\mathsf{e}\texttt{)} \\
\mathbf{i} & ::= & \cdots \mid {}^{\tau^v}\texttt{VS}(\mathsf{e}) \\
\mathbf{E} & ::= & \cdots \mid {}^{\tau^v}\texttt{VS}(\mathsf{E})
\end{array}$$

(a) Extensions to Fig. 2(a)

$$\begin{array}{lcl}
\mathsf{i} & ::= & \cdots \mid \texttt{SV}^{\tau^v}(\mathbf{e}) \qquad \mathsf{v} ::= \cdots \mid \texttt{bridge}\ \mathbf{m} \\
\mathsf{E} & ::= & \cdots \mid \texttt{SV}^{\tau^v}(\mathbf{E})
\end{array}$$

(b) Extensions to Fig. 2(b)

Fig. 3: Extended models for interoperability

A JavaScript expression e is a constant c, a variable x, variable binding, property load, property store, or property call. A constant is a number n, an object `[`A`]`, a first-class function `function` $(\overline{\mathsf{x}})$ $\{\mathsf{e}\}$, or `null`. We omit features like property deletion and prototypes that do not affect the interoperations. The operational semantics is conventional as well using object table $\mathcal{O}$ that maps object properties to their values.

B. Extended Models for Interoperability

To support interoperations, we extend each base model. Fig. 3(a) and Fig. 3(b) show extensions to the base Java and JavaScript models, respectively. They introduce additional expressions ${}^{\tau^v}\texttt{VS}(\mathsf{e})$ and $\texttt{SV}^{\tau^v}(\mathbf{e})$ to indicate switches of languages. ${}^{\tau^v}\texttt{VS}(\mathsf{e})$ denotes a Java boundary expression of type τ^v that contains JavaScript expression e. Similarly, $\texttt{SV}^{\tau^v}(\mathbf{e})$ denotes a JavaScript boundary expression that contains Java expression $\mathbf{e}$ of type τ^v. For indistinguishable interoperability, external syntax e does not include its cross-language boundary; $\texttt{SV}^{\tau^v}(\mathbf{e})$ is an internal expression i created at run time.

For dynamic interoperability, Fig. 3(a) specifies dynamic creation and destruction of a bridge using $\texttt{add}(\texttt{[}\mathbf{A}\texttt{]})$ and $\texttt{remove}(\texttt{[}\mathbf{A}\texttt{]})$, respectively: $\texttt{add}(\texttt{[}\mathbf{A}\texttt{]})$ enables access to a Java object $\texttt{[}\mathbf{A}\texttt{]}$ from JavaScript; and $\texttt{remove}(\texttt{[}\mathbf{A}\texttt{]})$ withdraws the accessibility of $\texttt{[}\mathbf{A}\texttt{]}$. Without loss of generality, we assume that JavaScript code uses a JavaScript object $\texttt{[}\mathsf{A}\texttt{]}$ to exploit a bridge for $\texttt{[}\mathbf{A}\texttt{]}$. We call such JavaScript objects *bridge objects*.

Another additional expression $\texttt{evaluate}(\mathsf{e})$ in Fig. 3(a) indicates two methods `loadUrl` and `evaluateJavascript` that Android provides to execute JavaScript code from Java code. Because both `loadUrl` and `evaluateJavascript` always return values of type `void`, the extended model initiates interoperations via $\texttt{evaluate}(\mathsf{e})$ instead of boundary ${}^{\tau^v}\texttt{VS}(\mathsf{e})$. Thus, Fig. 3(a) includes $\texttt{evaluate}(\mathsf{e})$ in external syntax $\mathbf{e}$ and hides ${}^{\tau^v}\texttt{VS}(\mathsf{e})$ under internal syntax $\mathbf{i}$.

Using this model, we can simplify Android interoperations. Consider the following Android Java code:

```
class A {
    @JavascriptInterface
    public int m(int x) { return x; }
}
WebView webView = ...;
webView.addJavascriptInterface(new A(), "a");
webView.evaluateJavascript("a.m(undefined);");
```

which injects an instance of class `A` into the JavaScript context with name `a`, and evaluates JavaScript code `a.m(undefined)` invoking Java method `m`. We encode it as follows in our model:

$$\begin{array}{l}\texttt{object}\ \mathbf{A}\ \{\ \iota\ \mathbf{m}(\iota\ \mathbf{x})\ \{\ \mathbf{x}\ \}\} \\ \texttt{add}(\texttt{[}\mathbf{A}\texttt{]}) \\ \texttt{evaluate}(\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}(\texttt{undefined}))\end{array}$$

We will describe reduction steps of such code in following subsections using context-sensitive reduction rules as with MF; we use an evaluation context $\mathbf{E}$ for Java and E for JavaScript. Also, we establish $\mathbf{E}$ as a top-level evaluation context $\mathcal{E}$ to indicate that the top-level program is written in Java.

Note that our model prohibits nested interoperability by separating bridging operations, $\texttt{add}(\texttt{[}\mathbf{A}\texttt{]})$, $\texttt{remove}(\texttt{[}\mathbf{A}\texttt{]})$, and $\texttt{evaluate}(\mathsf{e})$, from normal operations even though Android allows nested interoperations. However, our model still can represent the exact Android interoperation behaviors because of the asynchronous feature that we discuss in Section III-D.

C. Core Interoperation Semantics

We present the "core" interoperability semantics first, and then extend it to describe quirky behaviors that we discovered.

[Jv-Add] $$\frac{\mathcal{B}' = \mathcal{B} \cup \{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})\}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{add}(\texttt{[}\mathbf{A}\texttt{]})]_V \rightarrow \mathcal{B}'\ \mathcal{O}\ \mathcal{E}[\texttt{()}]}$$

[Jv-Remove] $$\frac{\mathcal{B}' = \mathcal{B} \backslash \{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})\}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{remove}(\texttt{[}\mathbf{A}\texttt{]})]_V \rightarrow \mathcal{B}'\ \mathcal{O}\ \mathcal{E}[\texttt{()}]}$$

[Jv-Evaluate] $$\frac{\dfrac{\mathcal{B} = \overline{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})}}{\texttt{object}\ \mathbf{A}\ \{\ \overline{\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}}\ \}} \qquad \mathcal{O}' = \overline{(\mathsf{A}.\mathsf{m}, \texttt{bridge}\ \mathbf{m})}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{evaluate}(\mathsf{e})]_V \rightarrow \mathcal{B}\ \mathcal{O}'\ \mathcal{E}[{}^{\texttt{void}}\texttt{VS}(\mathsf{e})]}$$

[Jv-VS-Void] $\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\texttt{void}}\texttt{VS}(\mathsf{v})]_V \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{()}]$

[Jv-VS-Num1] $\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\iota}\texttt{VS}(\mathsf{n})]_V \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\mathbf{n}]$

[Jv-VS-Num2] $$\frac{\mathsf{v} \neq \mathsf{n}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\iota}\texttt{VS}(\mathsf{v})]_V \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\mathbf{0}]}$$

[Jv-VS-Obj1] $$\frac{\mathcal{B}(\texttt{[}\mathsf{A}\texttt{]}) = \texttt{[}\mathbf{A}\texttt{]}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\mathbf{A}}\texttt{VS}(\texttt{[}\mathsf{A}\texttt{]})]_V \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathbf{A}\texttt{]}]}$$

[Jv-VS-Obj2] $$\frac{\mathsf{v} \notin \mathit{dom}(\mathcal{B})}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\mathbf{A}}\texttt{VS}(\mathsf{v})]_V \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{null}]}$$

[Jv-VS-Obj3] $$\frac{\mathcal{B}(\texttt{[}\mathsf{A}\texttt{]}) \neq \texttt{[}\mathbf{A}\texttt{]}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[{}^{\mathbf{A}}\texttt{VS}(\texttt{[}\mathsf{A}\texttt{]})]_V \rightsquigarrow \texttt{error"incompatible object"}}$$

(a) Extended Java semantics

[Js-JsCall] $$\frac{\mathcal{O}(\mathsf{A}.\mathsf{m}) = \texttt{function}\ (\overline{\mathsf{x}})\ \{\mathsf{e}\}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}(\overline{\mathsf{v}})]_S \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\mathsf{e}[\overline{\mathsf{v}/\mathsf{x}}]]}$$

[Js-JvCall1] $$\frac{\begin{array}{c}\mathcal{O}(\mathsf{A}.\mathsf{m}) = \texttt{bridge}\ \mathbf{m}' \qquad \mathcal{B}(\texttt{[}\mathsf{A}\texttt{]}) = \texttt{[}\mathbf{A}\texttt{]} \\ \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \qquad \tau^v\ \mathbf{m}'(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \in \{\overline{\mathbf{md}}\} \qquad |\overline{\mathbf{x}}| = |\overline{\mathsf{v}}|\end{array}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}(\overline{\mathsf{v}})]_S \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{SV}^{\tau^v}(\texttt{[}\mathbf{A}\texttt{]}.\mathbf{m}'(\overline{{}^{\tau^v}\texttt{VS}(\mathsf{v})}))]}$$

[Js-JvCall2] $$\frac{\begin{array}{c}\mathcal{O}(\mathsf{A}.\mathsf{m}) = \texttt{bridge}\ \mathbf{m}' \qquad \mathcal{B}(\texttt{[}\mathsf{A}\texttt{]}) = \texttt{[}\mathbf{A}\texttt{]} \\ \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \qquad \tau^v\ \mathbf{m}'(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \notin \{\overline{\mathbf{md}}\} \qquad |\overline{\mathbf{x}}| = |\overline{\mathsf{v}}|\end{array}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}(\overline{\mathsf{v}})]_S \rightsquigarrow \texttt{error "no bridge method"}}$$

[Js-JvCall3] $$\frac{\mathcal{O}(\mathsf{A}.\mathsf{m}) = \texttt{bridge}\ \mathbf{m}' \qquad \texttt{[}\mathsf{A}\texttt{]} \neq \mathit{dom}(\mathcal{B})}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}(\overline{\mathsf{v}})]_S \rightsquigarrow \texttt{error "non-bridge object"}}$$

[Js-SV-Void] $\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{SV}^{\texttt{void}}(\texttt{()})]_S \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{undefined}]$

[Js-SV-Num] $\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{SV}^{\iota}(\mathbf{n})]_S \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\mathsf{n}]$

[Js-SV-Obj1] $$\frac{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{SV}^{\mathbf{A}}(\texttt{[}\mathbf{A}\texttt{]})]_S \rightarrow \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}]}$$

[Js-SV-Obj2] $$\frac{\begin{array}{c}(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \notin \mathcal{B} \qquad \texttt{object}\ \mathbf{A}\ \{\ \overline{\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}}\ \} \\ \mathcal{B}' = \mathcal{B} \cup \{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})\} \qquad \mathcal{O}' = \mathcal{O} \cup \{\overline{(\mathsf{A}.\mathsf{m}, \texttt{bridge}\ \mathbf{m})}\}\end{array}}{\mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{SV}^{\mathbf{A}}(\texttt{[}\mathbf{A}\texttt{]})]_S \rightarrow \mathcal{B}'\ \mathcal{O}'\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}]}$$

(b) Extended JavaScript semantics

Fig. 4: Core interoperation semantics

Fig. 4(a) shows the semantics of bridge creation, destruction, and provision. Rules [Jv-Add] and [Jv-Remove] create and destruct bridges, respectively, by updating bridge table $\mathcal{B}$. Rule [Jv-Evaluate] provides available bridges in $\mathcal{B}$ to the JavaScript context by updating object table $\mathcal{O}$ with properties of bridge objects. A bridge object $\texttt{[}\mathsf{A}\texttt{]}$ contains methods of $\texttt{[}\mathbf{A}\texttt{]}$ as its properties: $\overline{(\mathsf{A}.\mathsf{m}, \texttt{bridge}\ \mathbf{m})}$ where $\texttt{bridge}\ \mathbf{m}$ denotes a Java method named $\mathbf{m}$ in the JavaScript context, called a *bridge method*. Fig. 4(b) shows the semantics of bridge objects and bridge methods, which intermingle with those of JavaScript expressions for indistinguishable interoperability. While we present the intermingled semantics in the Android hybrid mechanism in this paper, we can describe various interoperability semantics based on our formalization. For example, consider Java™ Plug-In [14] and Rhino [15] supporting communication between JavaScript code and Java applets running on a web page. While they have different semantics from that of Android, they also provide dynamic and indistinguishable communication between Java and JavaScript. Thanks to such common ground, we can use the Android interoperation semantics as a reference when specifying their semantics. As an example, we present an operational semantics of Java™ Plug-In in a companion report [16].

1) Method Overloading Resolution in JavaScript: While Java supports overloaded methods having the same name but with different signatures, JavaScript does not provide overloaded functions. Moreover, JavaScript allows any number of arguments for a function call regardless of the number of function parameters at its declaration site.

The Android hybrid mechanism partially supports overloaded Java methods; it selects a Java method to invoke from a list of available method declarations in a given Java object using the name and the arity of a method call. When there are multiple method declarations with the same name and the same arity but with different parameter types, it does not guarantee which method is called. When no method declarations have the same name and the arity of a given method call, it throws a "Method not found" exception. Rule [Js-JvCall1] in Fig. 4(b) specifies a successful Java method call from JavaScript code; calling a bridge method $\mathtt{bridge}\ \mathbf{m}'$ on a bridge object $\mathtt{[A]}$ with JavaScript values $\overline{\mathsf{v}}$. First, it searches for a Java method $\tau^v\ \mathbf{m}'(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}$ to invoke using the bridge method name $\mathbf{m}'$ and the number of arguments $|\overline{\mathsf{v}}|$. Then, it converts the arguments to Java values by $\overline{{}^{\tau^v}\mathsf{VS}(\mathsf{v})}$, invokes the Java method with the converted values, and converts the method call result via SV^{τ^v}. Rule [Js-JvCall2] specifies exception `"no bridge method"` when it cannot find the Java method. We use $\rightsquigarrow$ for exception-throwing rules to distinguish them from normal reduction rules.

2) First-Class Java Methods: While JavaScript functions are first-class values, Java methods are not. Java methods injected to the JavaScript context, which we call bridge methods, are considered as first-class values; we can assign them to variables or set them as object properties. However, they do not behave like normal JavaScript functions. For example, we can set a JavaScript object property with a bridge method `foo` of a bridge object `brg` as follows:

```
jsObj = {}
jsObj.bar = brg.foo;
```

but calling the property `jsObj.bar()` throws a "Java bridge method can't be invoked on a non-injected object" exception. Rule [Js-JvCall3] throws exception `"non-bridge object"` for bridge method calls on non-bridge (normal) objects.

3) Value Conversion between Java and JavaScript: Java and JavaScript have different values with different type systems. While Java provides user-defined types by statically declared classes, JavaScript does not support them but it allows users to construct and update objects at run time.

Android provides implicit value conversion between Java and JavaScript. When JavaScript code invokes a Java method with JavaScript values, Android converts the arguments to Java values of the method's parameter types. When values to be converted are inconsistent with given types, Android uses default values like $\mathbf{0}$ for a number and `null` for an object as shown in rules [Jv-VS-Num2] and [Jv-VS-Obj2] in Fig. 4(a). Rule [Jv-VS-Obj3] shows an unacceptable value conversion; converting a bridge object to a Java object of an incompatible type throws exception `"incompatible object"`. After invoking a Java method from JavaScript code, its result is converted to a JavaScript value. This conversion may implicitly inject Java objects to the JavaScript context. Consider rules [Js-SV-Obj1] and [Js-SV-Obj2] in Fig. 4(b) representing conversions from a Java object to a JavaScript object. If the Java object is already injected to the JavaScript context, Android uses the corresponding bridge object ([Js-SV-Obj1]). Otherwise, it dynamically injects the Java object to the JavaScript context on demand ([Js-SV-Obj2]).

D. Extensions for Wild Interoperation

Now, let us extend the core semantics to express wild Android interoperation behaviors.

1) Multiple Bridge Objects for Single Java Object: A single Java object can be bound to multiple bridge objects. For example, the following code:

```
1  Bridge obj = new Bridge();
2  addJavascriptInterface(obj, "b1");
3  addJavascriptInterface(obj, "b2");
4  loadUrl("file:///android_asset/www/index.html");
```

injects the same Java object `obj` into the JavaScript context with two different names `b1` and `b2` on lines 2 and 3. Because Android creates a new bridge object for a given name rather than for a given Java object, it maintains many-to-one mappings from bridge objects to Java objects. Thus, JavaScript code can make independent updates on `b1` and `b2`, which do not reflect to each other nor to the original Java object `obj`.

This functionality may be useful but it incurs unintuitive semantics. For instance, suppose that the above `obj` contains method `identity` that returns itself as it is, and that loading the above `index.html` executes the following JavaScript code:

```
b1.print = function() { console.log("b1"); };
b2.print = function() { console.log("b2"); };
b2.identity().print();
```

The code adds a property named `print` with a JavaScript function value to both bridge objects `b1` and `b2`. One may expect that `b2.identity()` would return `b2` as it is, thus `b2.identity().print()` would log `"b2"` to the console. However, unfortunately, it may log `"b1"` rather than `"b2"` because, when `b2.identity()` returns a Java object, Android does not guarantee to find the correct bridge object (`b2`) from multiple bridge objects (`b1` and `b2`). In short, Android uses many-to-one mappings from bridge objects to Java objects, which bring buggy results from Java method calls.

We can extend the core semantics to specify many-to-one mappings from bridge objects to Java objects. While the core semantics represents one-to-one mappings by the implicit correspondence between a Java object $\texttt{[}\mathbf{A}\texttt{]}$ and its bridge object $\texttt{[}\mathsf{A}\texttt{]}$, we can extend it to the correspondence between a Java object $\texttt{[}\mathbf{A}\texttt{]}$ and its multiple bridge objects $\{\overline{\texttt{[}\mathsf{A}\texttt{]}}\}$. Then, rule [Js-SV-Obj1] does not guarantee which bridge object among multiple ones would be returned.

2) Bridge Objects without Matching Java Objects: Android provides two methods `loadUrl` and `evaluateJavascript` to execute JavaScript code from Java code, but they provide bridges to the JavaScript context differently. `loadUrl` always provides *new* bridges while `evaluateJavascript` provides new bridges only when Android executes JavaScript code for the first time. With these new bridges, JavaScript code cannot access previous implicitly-injected Java objects. Also, `loadUrl` clears the JavaScript context except for its first call, but `evaluateJavascript` does not clear the JavaScript context. Therefore, sequentially invoking methods `evaluateJavascript` and `loadUrl` may lead to unintuitive behaviors. For example:

```
addJavascriptInterface(new Bridge(), "brg");
evaluateJavascript("var b = brg.create(); ... ");
loadUrl("file:///android_asset/www/index.html");
```

JavaScript code executed by `evaluateJavascript` implicitly injects a Java object as a result of `brg.create()`, but `loadUrl` disallows access to the injected object while preserving its bridge object `b`. In short, even though the Java object is no longer accessible from JavaScript, its corresponding bridge object is retained in the JavaScript context. Thus, if the JavaScript code in `index.html` invokes Java methods via `b`, it results in application crash.

We can describe such wild behaviors by classifying $\texttt{evaluate}(\mathsf{e})$ according to its behaviors and defining its semantics differently. We distinguish three kinds of $\texttt{evaluate}(\mathsf{e})$: 1) it provides *new* bridges to a *new* JavaScript context, which represents `evaluateJavascript` that executes JavaScript code for the first time and `loadUrl` after the first call, 2) it provides *new* bridges to the *existing* JavaScript context, which represents the first call of `loadUrl`, and 3) it provides the *existing* bridges to the *existing* JavaScript context. Rule [Jv-Evaluate] in Fig. 4(a) specifies the first case where a JavaScript expression is executed under newly constructed object table $\{\overline{(\mathsf{A.m}, \texttt{bridge}\ \mathbf{m})}\}$. The second case is as follows:

$$\frac{\begin{array}{c} \mathcal{B}_E = \{\overline{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})}\} \qquad \overline{\texttt{object}\ \mathbf{A}\ \{\ \overline{\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}}\ \}} \\ \{\overline{(\texttt{[}\mathsf{A}'\texttt{]}, \texttt{[}\mathbf{A}'\texttt{]})}\} \subseteq \mathcal{B} \qquad \{\overline{\texttt{[}\mathbf{A}'\texttt{]}}\} \cap \{\overline{\texttt{[}\mathbf{A}\texttt{]}}\} = \emptyset \\ \mathcal{B}' = \mathcal{B}_E \cup\ \{\overline{(\texttt{[}\mathsf{A}'\texttt{]}, \texttt{[]})}]\} \\ \mathcal{O}' = (\mathcal{O} \backslash \{\overline{(\mathsf{A.m}', \_)}\}) \cup \{\overline{(\mathsf{A.m}, \texttt{bridge}\ \mathbf{m})}\} \end{array}}{\mathcal{B}_E\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{evaluate}(\mathsf{e})]_V \rightarrow \mathcal{B}_E\ \mathcal{B}'\ \mathcal{O}'\ \mathcal{E}[{}^{\texttt{void}}\texttt{VS}(\mathsf{e})]}$$

By separately maintaining explicitly-injected Java objects in $\mathcal{B}_E$, it identifies previous implicitly-injected Java objects $\overline{\texttt{[}\mathbf{A}'\texttt{]}}$

$$\text{[Js-Cache1]}\ \frac{\texttt{[}\mathsf{A}\texttt{]} \notin \mathit{dom}(\mathcal{B})}{\mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}]_S \rightarrow \mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{access}(\texttt{[}\mathsf{A}\texttt{]}, \mathsf{m})]}$$

$$\text{[Js-Cache2]}\ \frac{\texttt{[}\mathsf{A}\texttt{]} \in \mathit{dom}(\mathcal{B}) \qquad (\mathsf{A.m}, \mathsf{JS}) \in \mathcal{C}}{\mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}]_S \rightarrow \mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{access}(\texttt{[}\mathsf{A}\texttt{]}, \mathsf{m})]}$$

$$\text{[Js-Cache3]}\ \frac{\texttt{[}\mathsf{A}\texttt{]} \in \mathit{dom}(\mathcal{B}) \qquad (\mathsf{A.m}, \mathsf{JV}) \in \mathcal{C}}{\mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}]_S \rightarrow \mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{bridge}\ \mathbf{m}]}$$

$$\text{[Js-Cache4]}\ \frac{\begin{array}{c} (\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B} \qquad \mathsf{A.m} \notin \mathit{dom}(\mathcal{C}) \qquad \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \\ \tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \in \{\overline{\mathbf{md}}\} \qquad \mathcal{C}' = \mathcal{C} \cup \{(\mathsf{A.m}, \mathsf{JV})\} \end{array}}{\mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}]_S \rightarrow \mathcal{C}'\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{bridge}\ \mathbf{m}]}$$

$$\text{[Js-Cache5]}\ \frac{\begin{array}{c} (\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B} \qquad \mathsf{A.m} \notin \mathit{dom}(\mathcal{C}) \qquad \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \\ \tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \notin \{\overline{\mathbf{md}}\} \qquad \mathcal{C}' = \mathcal{C} \cup \{(\mathsf{A.m}, \mathsf{JS})\} \end{array}}{\mathcal{C}\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}]_S \rightarrow \mathcal{C}'\ \mathcal{B}\ \mathcal{O}\ \mathcal{E}[\texttt{access}(\texttt{[}\mathsf{A}\texttt{]}, \mathsf{m})]}$$

Fig. 5: Cached property accesses of bridge objects

and disables accesses to them while retaining their bridge objects by mapping the bridge objects to dummy objects $\overline{\texttt{[]}}$. Note that the JavaScript code is executed with new bridges on top of the existing object table $\mathcal{O}$. Finally, for the third case, we use the existing object table as it is for the JavaScript code execution. Because the JavaScript context may contain bridge objects without corresponding Java objects, various unexpected behaviors may happen.

3) Cached Accesses of Bridge Object Properties: Bridge objects behave differently for different properties. Consider a bridge object `brg` for a Java object `obj`:

```
Bridge obj = new Bridge();
addJavascriptInterface(obj, "brg");
```

where `obj` contains a method named `foo`, but no methods named `bar`. It is okay to add a property named `bar` to `brg`, to modify it, and even to delete it from `brg` though such actions do not get reflected to `obj`. However, modifying or deleting properties whose names already exist in the corresponding Java object is silently ignored. Thus, JavaScript code `brg.foo = 4` evaluates to 4, but it does not change the value of `brg.foo` to 4. Moreover, reconsidering the example in Section III-D2:

```
addJavascriptInterface(new Bridge(), "brg");
evaluateJavascript("var b = brg.create(); ... ");
loadUrl("file:///android_asset/www/index.html");
```

One more quirky semantics exists here. If bridge object property `b.print` has a non-Java value such as a JavaScript function, evaluation of `b.print()` in `index.html` produces different results depending on whether `b.print` was accessed or not in the `...` part. This surprising behavior is due to the cached lookup process; because Android caches access of bridge object properties, `b.print` is still accessible in `index.html` if it was accessed before `loadUrl`.

Fig. 5 formally specifies cached access of bridge object properties in Android. Cache $\mathcal{C}$ records accessed properties of a bridge object; $\mathcal{C}(\mathsf{A.m}) = \mathsf{JS}$ denotes that property $\mathsf{A.m}$ was accessed and has a usual JavaScript value, and $\mathcal{C}(\mathsf{A.m}) = \mathsf{JV}$ denotes that property $\mathsf{A.m}$ was accessed and its value comes from the Java context. Rule [Js-Cache1] specifies that the property accesses of normal JavaScript objects behave the same, where `access([A],m)` follows rules [Js-Load1] and [Js-Load2] in Fig. 2(b). Rules [Js-Cache2] and [Js-Cache3] represent when a property of a bridge object has a cached JavaScript value and a cached bridge method, respectively. The last two rules are for cache misses, where the cache is extended with a new property information: (A.m,JV) or (A.m,JS). Using caches for bridge-object lookup may be efficient, but, given the unintuitive semantics, actual benefits may be questionable.

4) Asynchronous Boundaries: Android's `loadUrl` and `evaluateJavascript` execute JavaScript code asynchronously, making nested interoperations behave in an unintuitive manner. Consider Java class `NestedBr` having the following method:

```
@JavascriptInterface
public void inter() {
  evaluateJavascript(
    "brg.name = function() { console.log('brg') };");
}
```

and consider the following Java code:

```
  addJavascriptInterface(new NestedBr(), "brg")
  evaluateJavascript("brg.inter(); brg.name();")
```

One may expect that `brg.name()` would log `"brg"` to the console because call to `evaluateJavascript` in `brg.inter()` stores a log function to property `brg.name`. However, it throws a type error rather than printing `"brg"`. Because the `evaluateJavascript` method executes JavaScript code asynchronously, `brg.inter()` terminates without the property assignment; it updates the property after the previous `evaluateJavascript` call finishes. Thus, `brg.name()` raises an exception for calling the `undefined` value.

Since asynchronous boundaries do not affect the interoperation semantics besides the nested interoperation, we exclude them in this paper.

IV. Type System for Safe Hybrid Apps

In this section, as an application of the formal semantics, we define *safety* of Android interoperability and devise a type system that detects unsafe interoperations.

A. *Preventing the Bugs Detected by* HybriDroid

We define the safety as absence of error `MethodNotFound` that HybriDroid [7] detects from Android hybrid apps, and we design a type system detecting the `MethodNotFound` errors.

1) Safety: Safe interoperations involve no `MethodNotFound` errors defined by HybriDroid:

> When a bridge method call cannot find any target Java method to call, the bug detector reports error `MethodNotFound`, which may be due to three reasons. The first case is when a developer makes a typo in the bridge method name or indeed calls an undefined method of an injected object. The second case is when a target Java method is inaccessible either because it is private or it does not have the `@JavascriptInterface` annotation when the target Android version is 4.2 or later. Finally, the third case is when a bridge method call has a wrong number of arguments.

$$\text{[T-Jv-Add]}\quad \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{add}(\texttt{[}\mathbf{A}\texttt{]}) : \texttt{void}/\mathcal{B} \cup \{(\texttt{[A]},\texttt{[}\mathbf{A}\texttt{]})\}\ \tilde{\mathcal{O}}$$

$$\text{[T-Jv-Remove]}\quad \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{remove}(\texttt{[}\mathbf{A}\texttt{]}) : \texttt{void}/\mathcal{B}\backslash\{(\texttt{[A]},\texttt{[}\mathbf{A}\texttt{]})\}\ \tilde{\mathcal{O}}$$

$$\text{[T-Jv-Evaluate]}\quad \frac{\mathcal{B} = \overline{(\texttt{[A]},\texttt{[}\mathbf{A}\texttt{]})} \qquad \overline{\texttt{object}\ \mathbf{A}\ \{\ \overline{\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}}\ \}} \qquad \Gamma\ \mathcal{B}\ \overline{(\mathsf{A.m}, \mathsf{Fun}_V(\mathbf{m}))} \vdash \mathsf{e} : \tau^s/\mathcal{B}'\ \tilde{\mathcal{O}}'}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{evaluate}(\mathsf{e}) : \texttt{void}/\mathcal{B}'\ \tilde{\mathcal{O}}'}$$

$$\text{[T-Jv-VS]}\quad \frac{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \tau^s/\mathcal{B}'\ \tilde{\mathcal{O}}'}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash\ {}^{\tau^v}\texttt{VS}(\mathsf{e}) : \tau^v/\mathcal{B}'\ \tilde{\mathcal{O}}'}$$

Fig. 6: Type system for Java

The three cases represent possible failure scenarios in finding a corresponding Java method from a given property call on a bridge object. The first two cases stand for undefined property calls, and the third case indicates bridge method calls with a wrong number of arguments. Formally speaking:

Definition 1 (Safety): Top-level expression **e** is safe (in terms of interoperability) if for any $\mathcal{O}$ and $\mathcal{B}$

- $\mathbf{e} \neq \mathcal{E}[\texttt{[A]}.\mathsf{m}(\overline{\mathsf{v}})]_S$ where $(\texttt{[A]},\texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B}$, $\mathsf{A.m} \notin dom(\mathcal{O})$, and
- $\mathbf{e} \neq \mathcal{E}[\texttt{[A]}.\mathsf{m}(\overline{\mathsf{v}})]_S$ where $\overline{(\texttt{[A]},\texttt{[}\mathbf{A}\texttt{]})} \in \mathcal{B}$, $\mathcal{O}(\mathsf{A.m}) = \texttt{bridge}\ \mathbf{m}'$, $\overline{\texttt{object}\ \mathbf{A}\ \{\ \mathbf{md}\ \}}$, $\tau^v\ \mathbf{m}'(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \notin \{\overline{\mathbf{md}}\}$, $|\overline{\mathbf{x}}| = |\overline{\mathsf{v}}|$.

2) Type System: For simplicity, the type system detects only unsafe interoperations; it does not detect traditional type errors but extending it to detect them as well is straightforward. Java and JavaScript expressions share type environments; type table Γ maps variables to their types, bridge table $\mathcal{B}$ keeps bridges, and object table $\tilde{\mathcal{O}}$ maps object properties to their types.

Fig. 6 presents a type system snippet for Java; the rest rules are conventional. The full type system is available in a companion report [16]. Rules [T-Jv-Add] and [T-Jv-Remove] maintain bridges in $\mathcal{B}$. Rule [T-Jv-Evaluate] builds an object table based on $\mathcal{B}$ to type-check JavaScript code; it checks the type of e under a newly constructed object table $\overline{(\mathsf{A.m}, \mathsf{Fun}_V(\mathbf{m}))}$ where $\mathsf{Fun}_V(\mathbf{m})$ denotes a JavaScript type for Java methods named $\mathbf{m}$ in the JavaScript context. The typing rule for Java boundary ${}^{\tau^v}\texttt{VS}(\mathsf{e})$ checks the type of the wrapped JavaScript expression e. To focus on interoperability, we assume that Java objects and their method declarations are well-typed.

Fig. 7 presents a type system snippet for JavaScript. We introduce a simple type hierarchy with six kinds of JavaScript type τ^s: $\mathsf{Fun}_S(\overline{\mathsf{x}}, \mathsf{e})$ for JavaScript function, $\mathsf{Fun}_V(\mathbf{m})$ for bridge method `bridge` $\mathbf{m}$, $\mathsf{Obj}_S(\mathsf{A})$ for pure JavaScript object `[A]`, $\mathsf{Obj}_V(\mathsf{A}, \mathbf{A})$ for bridge object `[A]`, $\mathsf{Obj}(\mathsf{A})$ for encompassing $\mathsf{Obj}_S(\mathsf{A})$ and $\mathsf{Obj}_V(\mathsf{A}, \mathbf{A})$, and $\top$ for encompassing all JavaScript types. We use helper function `CheckTypes` to type-check a series of expressions.

Three typing rules for $\mathsf{e.m}(\overline{\mathsf{e}})$ in Fig. 7 check the type of the callee function e.m. If the type is $\mathsf{Fun}_S(\overline{\mathsf{x}}, \mathsf{e})$, it checks the argument expressions and the function body ([T-Js-JsCall]).

$$\tau^s ::= \mathsf{Fun}_S(\overline{\mathsf{x}}, \mathsf{e}) \ \mid\ \mathsf{Fun}_V(\mathbf{m}) \ \mid\ \mathsf{Obj}_S(\mathsf{A}) \ \mid\ \mathsf{Obj}_V(\mathsf{A}, \mathbf{A}) \ \mid\ \mathsf{Obj}(\mathsf{A}) \ \mid \top$$

$$\tau^s \ <:\ \top \qquad \mathsf{Obj}_S(\mathsf{A}) \ <:\ \mathsf{Obj}(\mathsf{A}) \qquad \mathsf{Obj}_V(\mathsf{A}, \mathbf{A}) \ <:\ \mathsf{Obj}(\mathsf{A})$$

$$\text{[T-Js-Fun]} \quad \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{function}(\overline{\mathsf{x}})\{\mathsf{e}\} : \mathsf{Fun}_S(\overline{\mathsf{x}}, \mathsf{e})/\mathcal{B}\ \tilde{\mathcal{O}}$$

$$\text{[T-Js-Bridge]} \quad \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{bridge}\ \mathbf{m} : \mathsf{Fun}_V(\mathbf{m})/\mathcal{B}\ \tilde{\mathcal{O}}$$

$$\text{[T-Js-JsObj]} \quad \frac{\texttt{[}\mathsf{A}\texttt{]} \notin \mathit{dom}(\mathcal{B})}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{[}\mathsf{A}\texttt{]} : \mathsf{Obj}_S(\mathsf{A})/\mathcal{B}\ \tilde{\mathcal{O}}}$$

$$\text{[T-Js-BrgObj]} \quad \frac{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{[}\mathsf{A}\texttt{]} : \mathsf{Obj}_V(\mathsf{A}, \mathbf{A})/\mathcal{B}\ \tilde{\mathcal{O}}}$$

$$\text{[T-Js-JsCall]} \quad \frac{\begin{array}{c} \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \mathsf{Obj}(\mathsf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}' \qquad (\mathsf{A}.\mathsf{m}, \mathsf{Fun}_S(\overline{\mathsf{x}}, \mathsf{e}')) \in \tilde{\mathcal{O}}' \\ \overline{\tau^s}\ \mathcal{B}''\ \tilde{\mathcal{O}}'' = \texttt{CheckTypes}(\Gamma\ \mathcal{B}'\ \tilde{\mathcal{O}}'\ \overline{\mathsf{e}}) \\ \Gamma \cup \{\overline{(\mathsf{x}, \tau^s)}\}\ \mathcal{B}''\ \tilde{\mathcal{O}}'' \vdash \mathsf{e}' : \tau^s/\mathcal{B}'''\ \tilde{\mathcal{O}}''' \end{array}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e}.\mathsf{m}(\overline{\mathsf{e}}) : \tau^s/\mathcal{B}'''\ \tilde{\mathcal{O}}'''}$$

$$\text{[T-Js-JvCall1]} \quad \frac{\begin{array}{c} \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \mathsf{Obj}_V(\mathsf{A}, \mathbf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}' \qquad (\mathsf{A}.\mathsf{m}, \mathsf{Fun}_V(\mathbf{m}')) \in \tilde{\mathcal{O}}' \\ \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \qquad \tau^v\ \mathbf{m}'(\overline{\tau^v\ \mathbf{x}})\{\mathbf{e}\} \in \{\ \overline{\mathbf{md}}\ \} \qquad |\overline{\mathbf{x}}| = |\overline{\mathsf{e}}| \\ \overline{\tau^v}\ \mathcal{B}''\ \tilde{\mathcal{O}}'' = \texttt{CheckTypes}(\Gamma\ \mathcal{B}'\tilde{\mathcal{O}}'\ \overline{{}^{\tau^v}\texttt{VS}(\mathsf{e})}) \\ \Gamma \cup \{(\overline{\mathbf{x}, \tau^v})\}\ \mathcal{B}''\ \tilde{\mathcal{O}}'' \vdash \texttt{SV}^{\tau^v}(\mathbf{e}) : \tau^s/\mathcal{B}'''\ \tilde{\mathcal{O}}''' \end{array}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e}.\mathsf{m}(\overline{\mathsf{e}}) : \tau^s/\mathcal{B}'''\ \tilde{\mathcal{O}}'''}$$

$$\text{[T-Js-JvCall2]} \quad \frac{\begin{array}{c} \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \mathsf{Obj}_S(\mathsf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}' \qquad (\mathsf{A}.\mathsf{m}, \mathsf{Fun}_V(\mathbf{m}')) \in \tilde{\mathcal{O}}' \\ \overline{\tau^s}\ \mathcal{B}''\ \tilde{\mathcal{O}}'' = \texttt{CheckTypes}(\Gamma\ \mathcal{B}'\tilde{\mathcal{O}}'\ \overline{\mathsf{e}}) \end{array}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e}.\mathsf{m}(\overline{\mathsf{e}}) : \top/\mathcal{B}''\ \tilde{\mathcal{O}}''}$$

$$\text{[T-Js-SV1]} \quad \frac{\tau^v \neq \mathbf{A}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{SV}^{\tau^v}(\mathbf{e}) : \top/\mathcal{B}\ \tilde{\mathcal{O}}}$$

$$\text{[T-Js-SV2]} \quad \frac{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \in \mathcal{B}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{SV}^{\mathbf{A}}(\mathbf{e}) : \mathsf{Obj}_V(\mathsf{A}, \mathbf{A})/\mathcal{B}\ \tilde{\mathcal{O}}}$$

$$\text{[T-Js-SV3]} \quad \frac{\begin{array}{c} (\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]}) \notin \mathcal{B} \qquad \texttt{object}\ \mathbf{A}\ \{\ \overline{\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\}}\ \} \\ \mathcal{B}' = \mathcal{B} \cup \{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})\} \qquad \tilde{\mathcal{O}}' = \tilde{\mathcal{O}} \cup \{\overline{(\mathsf{A}.\mathsf{m}, \mathsf{Fun}_V(\mathbf{m}))}\} \end{array}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \texttt{SV}^{\mathbf{A}}(\mathbf{e}) : \mathsf{Obj}_V(\mathsf{A}, \mathbf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}'}$$

Fig. 7: Type system for JavaScript

If the type of the callee is $\mathsf{Fun}_V(\mathbf{m})$, the callee is a bridge method named $\mathbf{m}$. When a receiver object is a bridge object, the corresponding Java object $\texttt{[}\mathbf{A}\texttt{]}$ should contain a declaration of method $\mathbf{m}$ ([T-Js-JvCall1]). Rule [T-Js-JvCall1] checks the types of the converted arguments $\overline{{}^{\tau^v}\texttt{VS}(\mathsf{e})}$ and of the converted result $\texttt{SV}^{\tau^v}(\mathbf{e})$. Note that typing rules [T-Js-JvCall1] and [T-Js-JvCall2] correspond to [Js-JvCall1] and [Js-JvCall3], respectively; there is no typing rule for [Js-JvCall2] denoting the `MethodNotFound` error. The typing rule for $\texttt{SV}^{\mathbf{A}}(\mathbf{e})$ with a not-yet-injected Java object $\texttt{[}\mathbf{A}\texttt{]}$ injects the Java object to the JavaScript context by introducing it to the object table ([T-Jv-SV3]).

Based on the formal semantics and the type system, we can show that a well-typed Android hybrid app does not contain any evaluation contexts that lead to error `MethodNotFound`. The full type system and proof using well-typed object tables [17] are available in a companion report [16].

$$\text{[T-Js-Store1]} \quad \frac{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \mathsf{Obj}_S(\mathsf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}' \qquad \Gamma\ \mathcal{B}'\ \tilde{\mathcal{O}}' \vdash \mathsf{e}' : \tau^s/\mathcal{B}''\ \tilde{\mathcal{O}}''}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e}.\mathsf{m} = \mathsf{e}' : \tau^s/\mathcal{B}''\ \tilde{\mathcal{O}}''[\mathsf{A}.\mathsf{m} := \tau^s]}$$

$$\text{[T-Js-Store2]} \quad \frac{\begin{array}{c} \Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e} : \mathsf{Obj}_V(\mathsf{A}, \mathbf{A})/\mathcal{B}'\ \tilde{\mathcal{O}}' \qquad \texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \} \\ \tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \notin \{\overline{\mathbf{md}}\} \qquad \Gamma\ \mathcal{B}'\ \tilde{\mathcal{O}}' \vdash \mathsf{e}' : \tau^s/\mathcal{B}''\ \tilde{\mathcal{O}}'' \end{array}}{\Gamma\ \mathcal{B}\ \tilde{\mathcal{O}} \vdash \mathsf{e}.\mathsf{m} = \mathsf{e}' : \tau^s/\mathcal{B}''\ \tilde{\mathcal{O}}''[\mathsf{A}.\mathsf{m} := \tau^s]}$$

Fig. 8: Extensions to Fig. 7

B. Extensions for Preventing Quirky Behaviors of Cached Property Access

We show how to extend the type system to prevent unintuitive behaviors discussed in Section III-D. We focus on quirks caused by cached access of bridge object properties (in Section III-D3) because the other quirks can be simply avoided; 1) multiple bridge objects for a Java object (in Section III-D1) can be syntactically checked and 2) bridge objects without corresponding Java objects (in Section III-D2) never happen in Android versions 7.0 and above. Note that we do not consider asynchronous boundaries (in Section III-D4) either because the asynchrony is orthogonal to the Android interoperation semantics.

1) Safety: Reconsider the quirky behaviors due to cached property access:

> Bridge objects behave differently for different properties. . . . However, modifying or deleting properties whose names already exist in the corresponding Java object is silently ignored.

Such ignored property assignments make property accesses to bridge objects difficult to understand, leading to programming errors. Thus, we extend the safety with *coherence* between assignments and accesses to bridge object properties; in safe Android hybrid apps, JavaScript changes bridge objects only with properties whose names do not overlap with those of the corresponding Java objects. We extend Definition 1 as follows:

Definition 2 (Safety): Top-level expressions $\mathbf{e}$ is safe if

- $\mathbf{e} \neq \mathcal{E}[\underline{\texttt{[}\mathsf{A}\texttt{]}.\mathsf{m}} = \mathsf{v}]_S$ where $\underline{(\texttt{[}\mathsf{A}\texttt{]}, \texttt{[}\mathbf{A}\texttt{]})} \in \mathcal{B}$, $\texttt{object}\ \mathbf{A}\ \{\ \overline{\mathbf{md}}\ \}$, $\tau^v\ \mathbf{m}(\overline{\tau^v\ \mathbf{x}})\ \{\mathbf{e}\} \in \{\overline{\mathbf{md}}\}$.

2) Type System: We now extend the type system in Section IV-A for the new safe interoperability. We change only the typing rules for property assignment as shown in Fig. 8. Property assignment to JavaScript objects can be conducted without any constraints ([T-Js-Store1]). On the other hand, assignment to bridge objects are constrained to use properties whose names do not exist in the corresponding Java objects ([T-Js-Store2]). We show that the extensions provide the new safe interoperability with the proof in a companion report [16].

V. Evaluation

In this section, we evaluate our implementation of the type system in terms of soundness, precision, and efficiency.

TABLE I: Interoperation bug detection results

Target Hybrid App	HybriDroid			Ours					
	#Bugs		Time (sec.)	#Bugs				Time (sec.)	Speed up
	FP	TP		FP due to HybriDroid	FP due to Rules	TP	Cause(#)		
`com.app.sophosi`	-	-	-	0	0	0		16.1	-
`com.appliclub1410`	-	-	-	0	1	0		181.4	-
`com.dhobilite`	-	-	-	0	1	0		13.6	-
`com.eclectictt`	-	-	-	0	0	0		18.0	-
`com.gameloft.android.ANMP.GloftA3HM`	0	0	54.3	1	0	1	Obfuscation(1)	2.1	25.9×
`com.gameloft.android.ANMP.GloftPEHM`	0	0	276.4	6	0	13	No Annotation(1) Obfuscation(12)	26.8	10.3×
`com.ibs.sketch`	0	0	5.5	1	0	0		1.6	3.4×
`com.lenovo.anyshare.cloneit`	0	1	42.8	0	0	1	No Method(1)	0.8	10.2×
`com.officeon_b`	-	-	-	0	0	0		24.5	-
`com.PrankFingerPregnancyTest2`	0	0	118.0	1	0	0		8.0	6.2×
`com.wirelesstag.android3`	-	-	-	0	3	0		35.8	-
`europe.de.ftdevelop.aviation.toolknife`	0	1	305.0	0	0	1	No Annotation(1)	7.9	38.6×
`mediavision.handwallet`	3	5	111.4	2	0	4	Obfuscation(4)	7.2	15.5×
`pozdravuha.ru.pozdravleniya`	0	1	36.4	0	0	1	Obfuscation(1)	2.2	16.5×
Total	**3**	**8**		**11**	**5**	**21**			

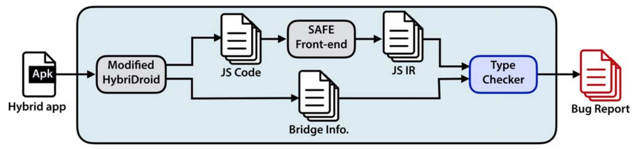

Fig. 9: Overview of the type system implementation

A. Implementation

We implemented our type system using two open-source tools, HybriDroid [7] and SAFE [18]. As rule [T-Jv-Evaluate] in Fig. 6 specifies, we need to collect bridges in bridge table $\mathcal{B}$ and to construct object table $\tilde{\mathcal{O}}$ in order to type-check JavaScript code. We use the front-end of HybriDroid to extract JavaScript code and its available bridges from Android hybrid apps. We modified HybriDroid to extract only necessary information without running its full analysis and to record this information in JSON format. We also use the front-end of SAFE to generate JavaScript intermediate representation that is suitable for type checking. Then, we implemented a type checker that constructs a bridge table and an object table from the recorded information and checks the type of the JavaScript code received from SAFE. Fig. 9 illustrates the overall architecture of the implementation. The source code of our tool is open to the public [19].

To address complex and extremely dynamic features of JavaScript, we extend the JavaScript typing rules in Fig. 7 and Fig. 8. First, we revise the simple type hierarchy of JavaScript to address functions as objects by making function types and bridge method types subtypes of object types. Second, we extend typing rules for function calls so that constructor calls, recursive calls, and unknown function calls have type $\top$, which may be imprecise but sound. Third, we extend typing rules for property read and write so that any accesses with non-literal property names have type $\top$. Finally, any language-constructs joining multiple values, such as conditional expressions and loops, have type $\top$. Even though our revised type system is aggressively losing precision, the experimental results in the next section show that it works well in practice.

B. Interoperation Bug Detection

To evaluate the soundness, precision, and efficiency of our type system, we compare its bug detection results with those of HybriDroid. The state-of-the-art static analyzer for Android hybrid apps, HybriDroid, conducts a costly flow-insensitive points-to analysis but with no formal ground, while our type system performs a light-weight linear traversal based on the formal interoperation semantics. For fair comparison, our experiments use the latest version of HybriDroid, which is much faster than its predecessor. We performed all the experiments on a Linux machine with 3.4GHz Intel Core i7 and 32GB Memory.

For analysis targets, we collected 231 real-world Android hybrid apps: 135 apps from PlayDrone [3] and 96 apps from AndroZoo [20]. Because HybriDroid collected hybrid apps from PlayDrone, we used PlayDrone for the fair comparison. In addition, because PlayDrone crawls apps from the 2014 Google Play store, we used AndroZoo as well to collect more up-to-date hybrid apps. From AndroZoo, we selected apps that are deployed more than 10 times in 2018 because they are likely to be well-maintained.

Table I summarizes the experimental results for AndroZoo dataset, especially for 14 hybrid apps that either HybriDroid or the type system fails to terminate or reports interoperation bugs for. The first column shows target apps, and the rest presents the numbers of false and true bugs, and the execution time in seconds for both HybriDroid and the type system. The execution time does not include the preparation time of HybriDroid, such as crawling JavaScript code to analyze, that is common for both HybriDroid and the type system. All figures in the table are averaged over 5 runs, and a dash (-) indicates a failure of analysis to terminate within 24 hours. We do not present the evaluation results for PlayDrone dataset in this paper because they show similar results to those for AndroZoo.

a) Soundness: The type checker detects 21 true interoperation bugs for the target subjects while HybriDroid reports only 8 true bugs. All the true bugs are `MethodNotFound` bugs caused by calling obfuscated methods with non-obfuscated names, methods not annotated with `@JavascriptInterface`, or undeclared methods. We reported these bugs to the corresponding developers. Note that the type checker did not find any bugs caused by cached property accesses. We manually investigated inter-language communication patterns in the target subjects and observed that developers use simple patterns defensively, which may explain why the type checker detected only `MethodNotFound` bugs. Also, the type checker finds all the true bugs detected by HybriDroid except one in `mediavision.handwallet`. We manually examined the missed true bug and found out that it is wrapped in an event callback registered by HTML DOM method `addEventListener`. Because the type system does not handle `addEventListener` unlike HybriDroid, it considers the buggy code unreachable. We believe that we can detect the bug when we introduce a model for event callbacks to the type system. The type checker also finds 14 more true bugs that HybriDroid fails to detect. We found a soundness bug in HybriDroid, confirmed by the authors.

b) Precision: The type checker reports 13 more false positives than HybriDroid, but only five of them are due to the type checking rules themselves as shown in Table I. We manually investigated them and found out that they are due to the imprecise type checking rule for dynamic property call `obj[prop](...)` where `prop` is not a literal. The type system simply gives type $\top$ for such dynamic property accesses and forbids property calls of type $\top$, which is sound but imprecise. The other 11 false positives are due to the incorrect bridge information extracted by the imprecise front-end of HybriDroid. While HybriDroid and our type system share the imprecise front-end, HybriDroid reports only two of 11 false positives in common. It does not report the rest nine false positives because of its soundness bug ironically; HybriDroid does not analyze some inter-language communications. We believe that we can remove the false positives by replacing the front-end of HybriDroid with a precise analyzer.

c) Efficiency: While HybriDroid takes 5.5 seconds to more than 24 hours in analyzing hybrid apps, the type checker takes 0.8 seconds to 181.4 seconds. As expected, the type checker detects interoperation bugs much faster than HybriDroid for all the subjects. The type checker shows the average speed-up of 15.8x compared to HybriDroid; we omitted the apps that HybriDroid fails to analyze within the time limit from the calculation of the average speed-up.

VI. Related Work

A. Analysis of Android hybrid apps

Researchers have revealed security vulnerabilities in Android hybrid apps, and developed analyzers that defend the apps against the vulnerabilities. [5] and [21] reported the security issue that no access control policy governs the Android interoperability, and proposed alternative policies. [4] reported new code injection attacks exploiting the Android interoperability, and they developed an analysis that detects potential injection flaws. While they propose defence mechanisms for security vulnerabilities utilizing the interoperability, we formalize the interoperation semantics.

Recent research has developed static analyzers that address control flows between Java and JavaScript. While [6] built control flow graphs of JavaScript and Java separately and added edges between them in a heuristic matter, HybriDroid [7] tracked flows seamlessly and detected interoperation bugs in real-world Android hybrid apps. Unlike HybriDroid, our type system is based on a formal semantics, which enables to uncover unintuitive, surprising interoperation behaviors, and to detect the bugs efficiently.

B. Semantics for multi-language programs

Researchers have studied multi-language semantics using Java™ Native Interface (JNI) [22]. Jeannie [23] is a language that integrates Java and C by compiling Jeannie to JNI, but its inter-language communication semantics are described informally using examples. JNI Light (JNIL) [24] formally describes interoperations between Java bytecode and assembly code. Similarly for JNIL, we formally describe interoperability in Java and JavaScript code, but our semantics is for high-level languages, in particular for typed and untyped languages.

Researchers also studied the interaction between typed and untyped languages. [10] used a syntactic *boundary* as a key construct for interoperations, and used a *guard* for the boundary to preserve type soundness. While [10] separated interoperations from normal operations via explicit syntactic boundaries, we do not distinguish them syntactically because the Android hybrid mechanism identifies such interoperations at run time. We also provide a way to control the interoperability granularity via *bridges*. [25] proposed a safe inheritance model between typed and untyped languages. We may apply the model to Android by creating JavaScript objects that inherit properties of Java classes, and by calling bridge methods using super calls. However, because the model prohibits bridge objects from using JavaScript's `Object` prototype, they may behave differently from original bridge objects.

VII. Conclusion

We presented a formal specification for inter-language communication in Android to provide a firm ground for understanding and reasoning about Android hybrid apps. We identified complex but under-documented interoperation behaviors via extensive testing and inspection of the Android source code. We extended the existing multi-language semantics to formally specify key interoperation features, dynamic and indistinguishable interoperations. Based on the extensions, we presented an operational semantics of the Android interoperability, and it enabled us to uncover quirky and error-prone behaviors in the Android hybrid mechanism. On top of the formalization, we designed and implemented a type system that detects interoperation bugs. We showed that it detects about twice more true bugs than HybriDroid with the 15.8x average speed-up. Moreover, our type system is just one application of the presented formal semantics. We believe one can design static analyzers or more expensive type systems based on the formalization in this paper.

Acknowledgment

This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

References

[1] I. Framework, “2017 developer survey,” https://www.ionicframework.com/survey/2017, 2017.

[2] OutSystems, “Mobility, custom apps a priority for 2015,” https://www.outsystems.com/1/mobility-custom-apps-report, 2015.

[3] N. Viennot, E. Garcia, and J. Nieh, “A measurement study of Google Play,” in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[4] 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*. ACM, 2014, pp. 66–77.

[5] 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*. ACM, 2014, pp. 1–8.

[6] A. D. Brucker and M. Herzberg, “On the static analysis of hybrid mobile apps,” in *International Symposium on Engineering Secure Software and Systems*, 2016.

[7] S. Lee, J. Dolby, and S. Ryu, “Hybridroid: Static analysis framework for android hybrid applications,” in *Proceedings of the 31th IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[8] Android, “Webview,” http://developer.android.com/reference/android/webkit/WebView.html, 2008.

[9] Google, “Chromium,” https://code.google.com/p/chromium/codesearch, 2016.

[10] J. Matthews and R. B. Findler, “Operational semantics for multi-language programs,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[11] A. Igarashi, B. C. Pierce, and P. Wadler, “Featherweight Java: A minimal core calculus for Java and GJ,” in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi, “The essence of JavaScript,” in *European Conference on Object-Oriented Programming*, 2010.

[13] M. Felleisen and R. Hieb, “The revised report on the syntactic theories of sequential control and state,” *Theoretical Computer Science*, vol. 103, no. 2, pp. 235–271, 1992.

[14] O. T. Network, “LiveConnect support in the next generation Java™Plug-In technology introduced in Java SE 6 update 10,” http://www.oracle.com/technetwork/java/javase/overview/liveconnect-docs-349790.html, 2006.

[15] M. D. Network, “Rhino,” https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino, 2005.

[16] S. Bae, S. Lee, and S. Ryu, “Towards understanding and reasoning about Android interoperations (supplementary material),” https://plrg.kaist.ac.kr/lib/exe/fetch.php?media=research:material:supplement.pdf, 2018.

[17] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[18] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, “SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript,” in *Proceedings of the International Workshop on Foundations of Object-Oriented Languages*, 2012.

[19] S. Bae, S. Lee, and S. Ryu, “Type system for safe interoperability in Android,” https://github.com/sorabae/android-typecheck, 2018.

[20] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon, “Androzoo: Collecting millions of android apps for the research community,” in *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016.

[21] G. Martin, S. Jana, and V. Shmatikov, “Breaking and fixing origin-based access control in hybrid web/mobile application frameworks,” in *NDSS symposium. Vol. 2014. NIH Public Access*, 2014.

[22] Oracle, “Java™ Native Interface,” http://docs.oracle.com/javase/7/docs/technotes/guides/jni/, 1999.

[23] M. Hirzel and R. Grimm, “Jeannie: Granting Java Native Interface developers their wishes,” in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2007.

[24] G. Tan, “JNI Light: An operational model for the core JNI,” *Journal of Mathematical Structures in Computer Science*, vol. 25, pp. 805–840, 2015.

[25] K. E. Gray, “Safe cross-language inheritance,” in *European Conference on Object-Oriented Programming*, 2008.