SAFE$_{\text{WAPI}}$: Web API Misuse Detector for Web Applications

SungGyeong Bae
KAIST, Korea
imai0917@kaist.ac.kr

Hyunghun Cho
Samsung Electronics, Korea
hyunghun.cho@samsung.com

Inho Lim
Samsung Electronics, Korea
inho0212.lim@samsung.com

Sukyoung Ryu
KAIST, Korea
sryu.cs@kaist.ac.kr

ABSTRACT

The evolution of Web 2.0 technologies makes web applications prevalent in various platforms including mobile devices and smart TVs. While one of the driving technologies of web applications is JavaScript, the extremely dynamic features of JavaScript make it very difficult to define and detect errors in JavaScript applications. The problem becomes more important and complicated for JavaScript *web* applications which may lead to severe security vulnerabilities. To help developers write safe JavaScript web applications using vendor-specific Web APIs, vendors specify their APIs often in Web IDL, which enables both API writers and users to communicate better by understanding the expected behaviors of the Web APIs.

In this paper, we present SAFE$_{\text{WAPI}}$, a tool to analyze Web APIs and JavaScript web applications that use the Web APIs and to detect possible misuses of Web APIs by the web applications. Even though the JavaScript language semantics allows to call a function defined with some parameters without any arguments, platform developers may require application writers to provide the exact number of arguments. Because the library functions in Web APIs expose their intended semantics clearly to web application developers unlike pure JavaScript functions, we can detect wrong uses of Web APIs precisely. For representative misuses of Web APIs defined by software quality assurance engineers, our SAFE$_{\text{WAPI}}$ detects such misuses in real-world JavaScript web applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Verification, Design

Keywords

JavaScript, static analysis, web application, bug detection

1. INTRODUCTION

Thanks to HTML5 [3] and web browsers on various platforms and devices, JavaScript [2] web applications are available everywhere. While JavaScript was originally developed as a simple scripting language to interact with users and to manipulate browser documents, it is now one of the most popular programming languages. The more diverse platforms and devices web applications support, the more complex and huge JavaScript web applications become.

The same reasons that have brought the popularity of JavaScript web applications also bring difficulties in developing them. Because HTML5 uses multiple languages including HTML, CSS, and JavaScript, web application developers should understand complex interactions between different languages. Also, to support diverse platforms and devices, web applications should interact with native applications or device functions seamlessly. While vendors provide Web APIs for JavaScript developers to use native and device functions, using such APIs correctly is a challenging task.

Indeed, many JavaScript errors originate from API function misuses. Ocariza *et al.* [25] manually inspected 12 bug repositories and reported that 74% of "fixed" JavaScript faults are from the "Incorrect Method Parameter" category, and 88% of the faults in the category are DOM-related. Their finding implies that 65% of JavaScript faults come from incorrect uses of DOM APIs by JavaScript code.

The main cause of the difficulties in using APIs correctly is the apparent gap between platform developers and web application developers. In order to use APIs correctly, web application developers should understand what platform developers intended when they designed APIs, which are often impossible because most API implementations are native or closed. Moreover, widespread uses of implicit control flows in API functions introduce another complication. For example, many API functions take callback functions as arguments from JavaScript developers and invoke them natively; a web application developer provides both a success callback and an error callback to an API function, and the API function invokes one of the callback functions depending on its execution result. We present an example of invoking callback functions within an API function in Section 2. Because API functions invoke callback functions with platform-generated objects as their arguments, faithfully testing such callback functions is almost impossible for JavaScript developers who cannot make platform-specific values.

To help JavaScript developers build correct web applications, researchers have studied static and dynamic analyses of JavaScript applications [15, 18, 19, 21, 11] but no existing tool can detect incorrect uses of API functions by JavaScript programs. JavaScript developers use JSLint [13] the most, but it detects only simple syntactic mistakes and violation of syntactic guidelines. Extremely dynamic semantics of JavaScript and complex interactions with embedding environments such as web browsers and device platforms make testing and analysis of JavaScript web applications sophisticated. Because the JavaScript errors defined in the language specification are too weak to find API misuses, and the dynamic features allow developers to intentionally write seemingly incorrect code, identifying API misuses is not crystal clear. Also, because the API implementations and platform-generated objects are not available to outside platforms, analyzing the interactions between JavaScript applications and APIs lacks important information.

In this paper, we present SAFE$_{\text{WAPI}}$, a static analyzer that detects API misuses in JavaScript web applications. To address the problems of analyzing JavaScript web applications, we utilize specifications of Web APIs often declared in interface description languages (IDLs) such as Web Interface Description Language (Web IDL) [7]. First, we use API specifications as criteria for defining API misuses. Even though the JavaScript language allows quirky semantics like providing no arguments to a function that takes multiple parameters, vendors describe their intention explicitly in API specifications requiring JavaScript developers to use APIs correctly by following the specifications. Second, we infer the important missing information from API specifications to connect control flows between web applications and API implementations. Because the specifications describe both what callback functions API functions expect and what platform-generated objects they use for callback functions precisely, we use such information to analyze interactions between JavaScript applications and Web APIs. Our SAFE$_{\text{WAPI}}$ collects information about API functions and objects from API specifications, models API functions and objects according to the specifications, and analyzes JavaScript web applications to detect misuses of APIs in them.

Our work includes the following contributions:

- **We provide a list of API misuses in JavaScript web applications in Section 3.** Even though defining errors in arbitrary JavaScript applications is not trivial, we can identify representative Web API misuses in JavaScript web applications with experienced software quality assurance engineers from industries. We illustrate how a JavaScript web application may use Web APIs incorrectly and how to detect such a misuse in Section 2.

- **We propose a mechanism to automatically model API functions and objects using API specifications provided by vendors in Section 4.** Supporting API functions and objects has been one of the most difficult problems for analyzers because their implementations are not available to analyze. While most analyzers require labor-intensive manual modeling of API functionalities, SAFE$_{\text{WAPI}}$ analyzes API specifications written in Web IDL and provides API modeling automatically. Our mechanism of automatic API modeling based on API specifications is not only for Web IDL and JavaScript but also directly applicable to other languages.

- **We describe our implementation of SAFE$_{\text{WAPI}}$ by extending a general analysis framework for JavaScript web applications, SAFE [4, 20], in Section 5.** SAFE$_{\text{WAPI}}$ first analyzes API specifications, uses the analyzed information to model APIs, and detects incorrect uses of APIs in JavaScript web applications. Using SAFE$_{\text{WAPI}}$, JavaScript programmers can develop and deploy safer JavaScript web applications on multiple platforms and devices to end-users by statically detecting errors before actually executing applications.

In Section 6, we describe how precisely and practically SAFE$_{\text{WAPI}}$ detects Web API misuses in various real-world JavaScript web applications. Section 7 discusses related work and we conclude in Section 8.

```
[NoInterfaceObject] interface CalendarManager {
  void getCalendars(CalendarType type,
                    CalendarArraySuccessCallback successCallback,
                    optional ErrorCallback? errorCallback);

  Calendar getUnifiedCalendar(CalendarType type);

  Calendar getDefaultCalendar(CalendarType type);

  Calendar getCalendar(CalendarType type, CalendarId id);
};
```

```
enum CalendarType { "EVENT", "TASK" };
```

```
[Callback=FunctionOnly, NoInterfaceObject]
interface CalendarArraySuccessCallback {
  void onsuccess(Calendar[] calendars);
};
```

```
[NoInterfaceObject] interface Calendar {

  readonly attribute CalendarId id;

  readonly attribute DOMString name;

  CalendarItem get(CalendarItemId id);

  void add(CalendarItem item);
```

Figure 1: **Web API specification written in Web IDL**

2. MOTIVATION

Before formally defining representative Web API misuses and how to detect them, we present a motivating example from industrial uses. Consider the following code:

```
function successCB(calendars) {
  calendars[0].foo;
}
var bar = "EVEN";
webapis.calendar.getCalendars(bar, successCB);
```

which calls `webapis.calendar.getCalendars`, a Samsung Web API [5] function. Figure 1 presents its specification written in Web IDL; it consumes two mandatory arguments of type `CalendarType` and `CalendarArraySuccessCallback`, and one optional argument of type `ErrorCallback`. If the API function call ends successfully, it invokes the second argument as a success callback, otherwise, it invokes the third argument, if any, as an error callback. Thus, if the above API call succeeds, it invokes `successCB`, whose type should be `CalendarArraySuccessCallback`, which is a function type that takes an array of the `Calendar` type as the Web API specification describes.

Let us take a closer look at the `successCB` callback. Because it has a function type that takes an array of `Calendar`, its parameter `calendars` should have the array of `Calendar` type. Then, the body of `successCB` accesses the 0th element of `calendars` by `calendars[0]`, which should have type `Calendar`. Finally, it refers the property `foo` of a `Calendar`, which does not have any property of name `foo`. Thus, we can detect that the body accesses a missing property, which leads to an undefined reference error.

This example illustrates two main benefits of using API specifications in our analysis. First, we can infer implicit control flows between JavaScript code and API functions from the information in API specifications. While a JavaScript developer who wrote the callback cannot see its invocation by the API function, we can infer with what type of value the API function invokes the callback function and analyze them by connecting their call flows. Second, we can use richer type information about JavaScript values using the specifications. For example, because `bar` in the above example is the first argument to the API function, it should have type `CalendarType`, which has only two possible values `"EVENT"` and

"TASK". With this type information, we can detect the error that bar has an invalid value "EVEN". We define representative Web API misuses based on real-world industrial experiences in Section 3, and we describe how we infer implicit control flows and how we detect type mismatches in Section 4.

3. WEB API MISUSES

The main observation of our analysis is that while the JavaScript language semantics permits wild uses of dynamic features, platform and device vendors who design and implement Web APIs for JavaScript developers strongly prefer to specify the intended semantics of API functions and require web application developers to follow the specifications. Based on our real-world experiences with designing industrial Web APIs and evaluating JavaScript web applications using Web APIs, we identify 6 common patterns of Web API misuses with software quality assurance engineers:

1. Accesses to absent properties of platform objects (AbsProp)
2. Wrong number of arguments to API function calls (ArgNum)
3. Missing error callback functions (ErrorCB)
4. Unhandled API calls that may throw exceptions (ExnHnd)
5. Wrong types of arguments to API function calls (ArgTyp)
6. Accesses to absent attributes of dictionary objects (AbsAttr)

3.1 Accesses to absent properties of platform objects

Frequent mistakes by JavaScript web applications are due to misunderstanding of *platform objects*. We call objects that are created by platforms and used by JavaScript applications via Web APIs platform objects. Similarly for JavaScript objects, platform objects may have various properties. While Web APIs specify a list of properties a platform object should provide, tracking what properties a given platform object provides is nontrivial. The example in Section 2 belongs to this pattern. The success callback function given as the second argument of the getCalendars function may be invoked by platform functions with platform objects bound to calendars. As we described with the example, detecting the erroneous access to the absent property foo of a Calendar object requires to keep track of various information often leading to programmer errors.

3.2 Wrong number of arguments to API function calls

While JavaScript allows to pass any number of arguments to any function calls, Web IDL can specify requirements on API function parameters. In JavaScript, even though a function definition takes a single parameter, it is perfectly fine not to provide any argument or to provide more than one argument. If the function is called without any argument, the parameter implicitly gets the undefined value, and if the function receives more than one argument, the second to the last arguments are evaluated but silently ignored. However, to help web applications use platform functions more reliably, Web APIs specify the minimum and the maximum numbers of arguments that an API function expects. For example, the Calendar Web API shown in Figure 1 specifies that the getCalendars function may take two or three parameters as follows:

```
void getCalendars(
  CalendarType type,
  CalendarArraySuccessCallback successCallback,
  optional ErrorCallback? errorCallback);
```

Thus, the function should take at least a calendar type and a success callback function but it may not receive an error callback function. If one invokes the function without any arguments, it throws a TypeError exception. Note that Web IDL supports optional parameters and function overloading unlike the JavaScript language, which make checking of valid number of arguments even more complicated.

3.3 Missing error callback functions

As we have already seen in previous examples, Web APIs extensively use callback functions to be responsive to user interactions. Many Web API functions take both success callback functions and error callback functions. If a Web API function executes successfully it calls its success callback function; otherwise, it calls its error callback function. Because such Web API functions execute asynchronously with JavaScript web applications, if a web application developer does not provide an error callback function to a Web API function, the developer may not know whether the API function performs as expected. Thus, missing (optional) error callback functions may not cause run-time errors, but providing error callback functions all the time is more defensive programming. For example, even though the Calendar Web API specifies that the third parameter of the getCalendars function is optional, because its type is ErrorCallback we detect the calls of getCalendars without the third argument as a warning. We made this decision because, in theory, the function call may silently ignore a failed execution of the function if it does not have any error callback function, and, in practice, missing error callback functions often degrades productivity of web application developers.

3.4 Unhandled API calls that may throw exceptions

Some API function calls may throw exceptions and Web APIs can describe such cases as follows:

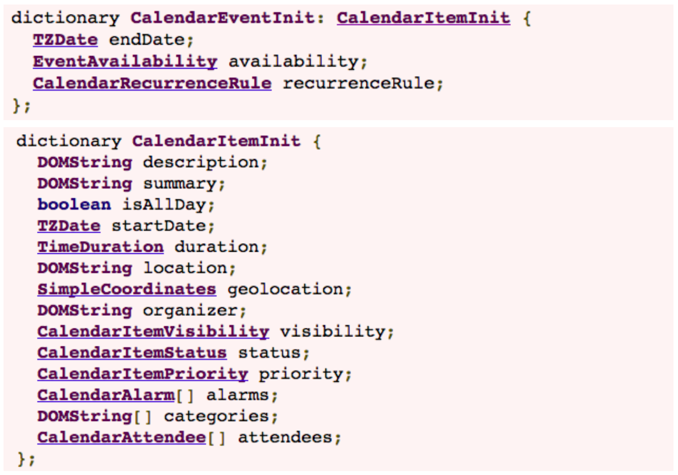
```
void getAddressBooks(
  AddressBookArraySuccessCallback successCallback,
  optional ErrorCallback? errorCallback
) raises(WebAPIException);
```

Because a call of the getAddressBooks function may throw the WebAPIException, JavaScript web applications that use the function may abruptly terminate due to uncaught exceptions. Thus, web applications should wrap any invocation of Web API functions that may throw exceptions with the try statement. For example, when one invokes getAddressBooks in a JavaScript code, it should defend the code as follows:

```
try {
  webapis.contact.getAddressBooks(succCB, errCB);
} catch (error) {
  console.log(error.name);
}
```

3.5 Wrong types of arguments to API function calls

While JavaScript function definitions do not specify the expected types of parameters, Web IDL specifies diverse properties including types of API function parameters. In JavaScript, functions may receive any values or even no values as arguments at function call sites. Moreover, due to the dynamic nature of JavaScript, the ECMAScript language specification [2] assigns one chapter to describe implicit conversions between various types, which limit possibilities of run-time type mismatches. Except for critical situations



```
dictionary CalendarEventInit: CalendarItemInit {
  TZDate endDate;
  EventAvailability availability;
  CalendarRecurrenceRule recurrenceRule;
};
```

```
dictionary CalendarItemInit {
  DOMString description;
  DOMString summary;
  boolean isAllDay;
  TZDate startDate;
  TimeDuration duration;
  DOMString location;
  SimpleCoordinates geolocation;
  DOMString organizer;
  CalendarItemVisibility visibility;
  CalendarItemStatus status;
  CalendarItemPriority priority;
  CalendarAlarm[] alarms;
  DOMString[] categories;
  CalendarAttendee[] attendees;
};
```

Figure 2: **Dictionary type definitions in Web IDL**

such as calling non-function objects, JavaScript code seldom terminates abnormally. However, to develop more reliable web applications, Web APIs specify the expected parameter types of API functions and they require web applications to satisfy the type requirements. For example, because `getCalendars` expects a value of type `CalendarType` as its first argument, the function should receive either `"EVENT"` or `"TASK"` as its first input. Because the example in Section 2 receives an invalid value `"EVEN"` for the first argument, which is not of type `CalendarType`, the Web API function throws a `TypeMismatchError` exception.

3.6 Accesses to absent attributes of dictionary objects

Unlike JavaScript that has only simple primitive types and object types without any user-defined types, Web IDL provides richer types including dictionary types [7]:

> “A dictionary is a definition (matching Dictionary) used to define an associative array data type with a fixed, ordered set of key-value pairs, termed dictionary members, where keys are strings and values are of a particular type specified in the definition.”

Dictionary type values in Web IDL correspond to JavaScript object values where object properties correspond to dictionary members. In order for a JavaScript object to satisfy a dictionary type defined in Web IDL, all the properties of the JavaScript object should be the members of the dictionary type. If the object has a property that is not a member of the dictionary type, the object does not have the dictionary type.

For example, consider the definitions of two dictionaries in Figure 2. The definition of the `CalendarEventInit` dictionary specifies that a dictionary of type `CalendarEventInit` may have members including `endDate`, `availability`, and `recurrenceRule` and it may inherit members from the `CalendarItemInit` dictionary, which has 14 more members. Therefore, while JavaScript objects such as `{isAllDay:true,availability:"FREE"}` and `{availability:"BUSY"}` have type `CalendarEvenetInit`, a JavaScript object `{foo:0}` does not. Thus, when a JavaScript application calls a constructor that expects an input value of type `CalendarEventInit`, the input JavaScript object should have only the properties that are members of the type. Thus, when a constructor `CalendarEvent` expects a value of type `CalendarEventInit`, the constructor call below is valid:

```
var ev = new webapis.CalendarEvent(
        {description: 'HTML5 Introduction',
         summary    : 'HTML5 Webinar',
         startDate  : new webapis.TZDate(
                          2011, 3, 30, 10, 0),
         duration   : new webapis.TimeDuration(
                          1, "HOURS"),
         location   : 'Huesca'})
```

but an invalid constructor call with a wrong input value such as `new webapis.CalendarEvent({foo:0})` may fail silently without producing an expected result.

4. TECHNICAL DETAILS

In this section, we describe how we fill in the missing information about Web API functions using their specifications. In order to analyze JavaScript web applications, our analysis should analyze both explicit and implicit execution flows between JavaScript programs and API functions. Even though we can analyze JavaScript programs because we have their source code, we cannot analyze API functions because either their sources are closed or their implementations are in different languages. Thus, to analyze API function calls in JavaScript web applications, we analyze their execution flows as follows:

1. To analyze explicit execution flows from JavaScript code to API functions, we check whether the JavaScript argument values to API functions satisfy the types of the corresponding parameters declared in the API specifications.

2. To analyze explicit execution flows from API functions to JavaScript code, we check whether JavaScript code uses the return values of API function calls correctly.

3. To analyze implicit execution flows hidden inside API functions, we check whether JavaScript callback functions use platform-generated objects correctly.

We briefly introduce our analysis based on abstract interpretation in Section 4.1. Then, we describe how we check the types of argument values to API functions in Section 4.2, how we model platform-generated objects to represent returned values from API function calls in Section 4.3, and how we model API function calls that invoke callback functions in Section 4.4.

4.1 Type-based analysis for JavaScript

Our tool to analyze JavaScript web applications builds on top of an existing static analysis framework that detects type-related errors in JavaScript programs [20]. The design and implementation of the analysis are in the abstract interpretation framework, and both the formal specification and the analysis implementation are open to the public [4]. While the analysis framework provides a simple abstract domain and a type-based analysis engine by default, the framework design is pluggable in the sense that researchers can replace the default abstract domain or the default analyzer with their own artifacts.

We describe the implementation of the analysis framework briefly in Section 5.1. Due to the space limitation, we omit the details of the analysis framework and refer the interested readers to the literature [20].

4.2 JavaScript values and Web IDL types

To check whether a Web API function receives valid JavaScript values as its arguments, we should check whether the values satisfy the corresponding parameter types in Web IDL. While JavaScript

Table 1: **Type conversion between Web IDL types and JavaScript values**

Argument	Integer	Float	DOMString	Boolean	Enum	Array	Dictionary	Callback	Interface
`undefined`	`0`	`Error`	`"undefined"`	`false`		`Error`	`Error`	`Error`	`Error`
`null`	`0`	`0.0`	`"null"`	`false`		`Error`	`Error`	`Error`	`Error`
`true`	`1`	`1.0`	`"true"`	`true`		`Error`	`Error`	`Error`	`Error`
`false`	`0`	`0.0`	`"false"`	`false`		`Error`	`Error`	`Error`	`Error`
`""`	`0`	`0.0`	`""`	`false`		`Error`	`Error`	`Error`	`Error`
`"1.2"`	`1`	`1.2`	`"1.2"`	`true`		`Error`	`Error`	`Error`	`Error`
`"one"`	`0`	`Error`	`"one"`	`true`		`Error`	`Error`	`Error`	`Error`
`0`	`0`	`0.0`	`"0"`	`false`		`Error`	`Error`	`Error`	`Error`
`Infinity`	`0`	`Error`	`"Infinity"`	`true`		`Error`	`Error`	`Error`	`Error`
`42`	`42`	`42.0`	`"42"`	`true`		`Error`	`Error`	`Error`	`Error`
`NaN`	`0`	`Error`	`"NaN"`	`false`		`Error`	`Error`	`Error`	`Error`
object			`"[object Object]"`	`true`		`Error`	object	`Error`	
`[9]`	`9`	`9.0`	`"9"`	`true`		`[9]`	`Error`	`Error`	`Error`
array	`0`	`Error`		`true`		array	`Error`	`Error`	
function	`0`	`Error`		`true`		`Error`			

```
[NoInterfaceObject] interface File {
  readonly attribute File? parent;

  readonly attribute boolean readOnly;

  readonly attribute boolean isFile;

  readonly attribute boolean isDirectory;

  readonly attribute Date? created;

  readonly attribute Date? modified;

  readonly attribute DOMString path;

  readonly attribute DOMString name;

  readonly attribute DOMString fullPath;

  readonly attribute unsigned long long fileSize;

  readonly attribute long length;

  DOMString toURI() raises(WebAPIException);
```

Figure 3: **File type definition in Web IDL**

has only 5 primitive types—undefined, null, boolean, string, and number—and object types, Web IDL provides rich types. Moreover, we should also consider a large set of implicit type conversion rules in JavaScript. Table 1 summarizes the conversions rules between JavaScript values and Web IDL types.

For a given JavaScript value on the first column, Table 1 shows whether the value satisfies the Web IDL type denoted by the second to the last columns; it assumes that the value satisfies the nullable and optional attribute of the type specification. The table illustrates most representative cases and refers the interested readers to the Web IDL specification for more complex cases denoted by empty gray cells. For example, the `undefined` value implicitly converts to `0` in a type context expecting an integer, it implicitly converts to `"undefined"` in a context expecting a DOMString, and it converts to `false` in a boolean context. The Web IDL specification describes how to check `undefined` in an enumeration type context: if `"undefined"` is not a value of the enumeration type, it throws an exception. Finally, the `undefined` value throws an exception for the other cases.

4.3 Automatic modeling of platform-generated objects

To check whether JavaScript web applications use return values from API function calls correctly, we automatically model platform-generated objects to represent such return values from API calls, and we use them as abstract values to analyze the JavaScript applications. Because API specifications include all the type definitions and the types of API functions, we can automatically construct a simple but valid object for each type defined in API specifications. For example, consider that we analyze a JavaScript program which calls a Web API function, receives a result from the call, and uses the result of type `File` shown in Figure 3. The definition of `File` declares that any value of type `File` should have several properties including `parent` of type `File`, `readOnly` of type `boolean`, and `created` of type `Date`. Based on the type definition, we can generate an object of type `File` accordingly and use it as an analysis value for the API function result. Note that constructing such objects for the types defined in API specifications may be recursive. Because the `parent` property of a `File` object also has type `File`, when we generate a `File` object, we need another `File` object as the value of the object's `parent` property.

Therefore, we automatically generate one representative platform-generated object, which we call *an abstract type value*, for each type in API specifications in advance. For each type defined in API specifications, we construct its abstract type value as follows:

- If it is one of the primitive types, undefined, null, boolean, string, and number, the corresponding abstract type value is the top abstract value of that type domain.
- If it is a composite type, the corresponding abstract type value has every property described in the API specification, and each property has its corresponding abstract type value.
- If it is a function type, the corresponding abstract type value is an abstract function object that returns an abstract type value that corresponds to the return type of the function type.

Using generated abstract type values for the types defined in API specifications, we can analyze explicit execution flows from API functions to JavaScript code so that we can check whether JavaScript programs use platform-generated objects correctly. Because abstract type values are sound approximations of concrete platform-generated objects, we add abstract operations on abstract type values that also work as sound approximations of their corresponding concrete operations. For example, when JavaScript code compares platform-generated objects using comparison operators like == or !==, we use a sound abstraction of the operators, which always returns the top abstract value of the boolean domain.

4.4 Automatic modeling of API functions

Finally, to analyze implicit execution flows between JavaScript code and API functions via callback function calls, we model API function calls that invoke callback functions automatically. Because JavaScript functions and API functions are usually in different programming models, they often communicate with each other by asynchronous callback function calls. While JavaScript developers write callback functions that API functions may invoke during their execution, JavaScript web applications do not have explicit callsites of callback functions. However, we can safely assume that API functions that take callback functions as arguments may invoke the input callback functions.

Thus, we model such implicit execution flows by analyzing hidden callback function calls with abstract type values. From a given JavaScript web application, we identify API function calls that take callback functions using the API specifications, and we add imaginary calls of the input callback functions. For example, consider the same example code from Section 2:

```
function successCB(calendars) {
  calendars[0].foo;
}
var bar = "EVEN";
webapis.calendar.getCalendars(bar, successCB);
```

We can easily identify that the API function call of `getCalendars` takes a callback function `successCB` as its second argument. Then, our analysis adds an imaginary call of `successCB` inside the invisible body of the API function. To make a callback function call, we should make its argument values, if any. Even though JavaScript callback functions do not specify the expected types of their parameters, we can get that information from API specifications. For example, as we discussed in Section 2, the API specification helps us find out that the callback takes an array of `Calendar`. Thus, we use an abstract type value corresponding to the array of `Calendar` type to invoke the callback function.

Note that an API function may take multiple callback functions as arguments and invoke one of them depending on the execution result of the API function. Indeed, the `getCalendars` API function has an optional callback function parameter to invoke when the execution of the API function terminates abnormally. The second parameter is a callback function to invoke when the API function execution terminates normally.

To safely model such API functions, our analysis adds imaginary calls of all the callback functions, and joins the analysis results from all the imaginary calls. Roughly speaking, our analysis models an API function `API` that takes non-callback parameters `arg_1`, . . . and callback parameters `CB_1, ..., CB_n` as follows:

```
function API(arg_1, ..., CB_1, ..., CB_n) {
  ... // its body without the result
  if (*) {
    CB_1(arg_11, ...);
  } else if (*) {
    CB_2(arg_21, ...);
  } .
    .
    .
  } else if (*) {
    CB_n(arg_n1, ...);
  } else {
    // when it does not call any callbacks
  }
  ... // return the result
}
```

It invokes all the callback functions with their corresponding abstract type values as arguments; for example, it invokes the `n`th callback function `CB_n` with abstract type value `arg_n1, ...` corresponding to the parameter types of `CB_n`. The last branch denotes the case when the API function does not invoke any callback functions. The order of imaginary callback function calls does not affect the analysis results because an API function invokes at most one callback function.

5. IMPLEMENTATION

Now, we describe our implementation of SAFE$_{\text{WAPI}}$, a static analyzer that detects various misuses of Web APIs in JavaScript web applications. Based on our experiences with real-world industrial applications, we use the following domain-specific observations:

- The root platform object of the Web APIs, `webapis`, is available from the global environment so that JavaScript developers can use Web APIs as property accesses of the object like `webapis.calendar.getCalendar`. The `webapis` object includes all the Web APIs information, and it is not extensible.
- JavaScript applications use Web APIs only as property accesses of `webapis`; they do not assign Web API features to JavaScript variables.

Though general JavaScript programs may not satisfy the requirements, all the real-world JavaScript web applications we collected to analyze satisfy them as we describe in Section 6. Even when JavaScript web applications do not satisfy the requirements, we can adjust our analysis to identify the root platform object of Web APIs and the callsites of Web API functions. For presentation brevity, we assume that the above requirements hold for our target JavaScript web applications.

5.1 SAFE: type-based analysis for JavaScript

We build SAFE$_{\text{WAPI}}$ by extending SAFE [20], a scalable analysis framework for JavaScript applications. Because detecting misuses of Web APIs requires a deep semantic analysis to infer argument values of Web API functions, for example, we extend Analyzer, a default type analyzer of SAFE. The SAFE framework is pluggable in the sense that one can replace the default analyzer with an alternative analyzer. To make the framework as scalable and pluggable as possible, the SAFE architecture shown inside the box of Figure 4 clearly separates concerns between components by well-defined interfaces, and their formal specification and implementation are available to the public [4].

SAFE can analyze both stand-alone JavaScript applications and web applications including a collection of JavaScript programs. Figure 4 describes a more general scenario of analyzing web applications using JavaScript programs, where dashed boxes denote data and solid boxes denote phases that manipulate and convert them. SAFE takes an HTML document or a directory to analyze, collects all the JavaScript codes either imported to HTML documents or embedded in them, converts them into CFGs (Control Flow Graphs) via conventional compilation steps, and finally analyzes the JavaScript programs in CFGs. To analyze web applications precisely, the analyzer should know various information such as the enclosing HTML documents, DOM (Document Object Model) APIs to access them, and JavaScript built-in objects and functions. SAFE builds DOM trees by parsing HTML documents, and constructs an initial heap that contains the global information to analyze web applications using DOM trees via Heap Builder. For a given program, the analyzer infers types for the expressions in the program, and reports possible bugs in them by Bug Detector.

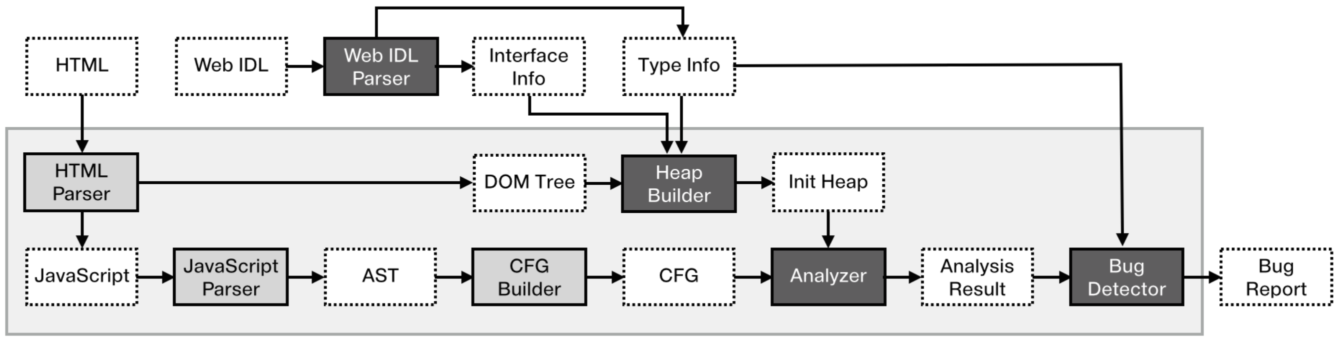


Figure 4: **Architecture of SAFE$_{\text{WAPI}}$**

5.2 SAFE$_{\text{WAPI}}$: SAFE with analysis of Web APIs

To analyze web applications using Web APIs correctly, we extend SAFE to understand Web APIs written in Web IDL. As the dark boxes and data around them in Figure 4 illustrate, we add Web IDL Parser that parses and analyzes Web APIs to build a richer initial heap by extended Heap Builder so that the extended Analyzer and Bug Detector can report misuses of Web APIs.

First, Web IDL Parser parses a given API specification written in Web IDL and extracts necessary information to analyze API functions: interfaces and types. While interfaces in Web APIs correspond to JavaScript objects and functions, Web IDL types do not correspond to anything in JavaScript but describe expected behaviors of API functions.

Second, the extended Heap Builder builds an initial heap using the additional interface and type information from Web APIs. It adds the root platform object, `webapis`, that contains all the interface objects constructed by Web IDL Parser to the initial heap. Even though the Web IDL types do not correspond to any values in JavaScript, we extend the initial heap with abstract type values corresponding to Web IDL types. As we described in Section 4.3, we model all the Web IDL types with their abstract type values so that we can analyze API functions precisely.

Then, the extended Analyzer analyzes Web API function calls by using their modeled functions with abstract type values from the initial heap. Similarly for SAFE to model built-in functions and DOM API functions, we model Web API functions automatically from their API specifications. While modeling of pure JavaScript API functions that do not have any type information is often imprecise, which in turn degrades the analysis precision, modeling of Web API functions is precise enough to return abstract type values, which improves the analysis precision.

Finally, the extended Bug Detector uses the results of Analyzer to detect any occurrences of the Web API misuse patterns defined in Section 3. Note that all the misuse patterns except for AbsProp involve API function calls, which implies that the complex interactions between JavaScript code and API functions are vulnerable to JavaScript developer errors. While detecting some misuse patterns like ArgNum, ErrorCB, and ExnHnd requires only syntactic checks, detecting other misuse patterns—AbsProp, ArgTyp, and AbsAttr—require a deep semantic analysis. To detect the AbsProp misuse pattern, Bug Detector uses abstract type values constructed as described in Section 4.3 and modeling of API function calls explained in Section 4.4. To detect the ArgTyp and AbsAttr misuse patterns, Bug Detector checks whether the analysis results of API function arguments satisfy the type requirements declared in API specifications according to the type conversion rules in Table 1 presented in Section 4.2.

6. EVALUATION

Table 2: **Real-world JavaScript web applications that include Web API misuses**

App. Id	AbsProp	ArgNum	ErrorCB	ExnHnd	ArgTyp	AbsAttr	Total
tv-1	**1**	0	0	0	0	0	**1**
tv-4	0	0	1	11	0	0	**12**
tv-8	1	0	0	1	0	0	**2**
tv-15	1	0	0	0	0	0	**1**
tv-17	0	0	1	0	0	0	**1**
tv-19	0	0	1	0	0	0	**1**
tv-20	2	0	0	0	0	0	**2**
mb-2	0	0	0	2	0	0	**2**
mb-3	0	0	0	2	0	0	**2**
mb-5	0	0	0	1	0	0	**1**
mb-6	0	**2**	0	51	0	0	**53**
mb-7	0	0	0	4	0	0	**4**
mb-12	0	0	0	1	0	0	**1**
mb-15	0	0	0	0	0	0	**0**
mb-19	4	0	0	3	0	0	**7**
mb-20	0	0	**1**	**10**	0	0	**11**
Total	**9**	**2**	**4**	**86**	**0**	**0**	**101**

We evaluated SAFE$_{\text{WAPI}}$ with real-world web applications and test cases. In this section, we describe representative examples that the tool found precisely for the Web API misuse patterns described in Section 3.

6.1 Overview

We collected 43 JavaScript web applications that use Samsung Web APIs and ran SAFE$_{\text{WAPI}}$ to detect Web API misuses in the applications. While the applications are not yet open to the public, some of them will be available at the Samsung developers site [5] in the near future. The 43 web applications consist of 23 Smart TV applications and 20 Android mobile web applications; we call the nth Smart TV application tv-n and the nth Android mobile web application mb-n.

Even though all the web applications are products ready for deployment, we found Web API misuses from 7 TV applications and 9 mobile applications, and our colleague software quality assurance engineers confirmed the detected misuses. As Table 2 summarizes, we found 4 misuse patterns out of 6 from the real-world web applications. The majority of the detected misuses are unhandled exceptions from API function calls that may throw exceptions (ExnHnd), and about 10 % of the detected misuses are accessing absent properties by using deprecated API functions (AbsProp). While we did not detect 2 misuse cases (ArgTyp and AbsAttr) from the collected web applications that the development of which has finished, we expect to detect them from web applications in development because we detected them from various test cases. We elaborate each misuse case with concrete examples highlighted in Table 2.

6.2 Accesses to absent properties of platform objects

We found this misuse pattern in 4 TV applications and 1 mobile application. Consider the following simplified excerpt from tv-1:

```
var nserviceModule = {
  ...
  nservice.registerManagerCallback(
    nserviceModule.onDeviceStatusChange);
  ...
  onDeviceStatusChange: function(sParam){
    ...
    alert("[nServiceModule.onDeviceStatusChange]"+
          ": eventType = "+sParam.eventType+
          ", deviceName = "+sParam.deviceName+
          ", device Type = "+sParam.deviceType);
```

from which $\mathsf{SAFE}_{\mathsf{WAPI}}$ detects a misuse as follows:

```
nserviceModule.js:21:149~21:166: [Warning]
  Reading absent property 'deviceType' of object
  'sParam'.
```

Note that detecting such a misuse requires understanding of the Web APIs and tracking of the information flow between JavaScript code and platform code via asynchronous callback functions. To see why `sParam.deviceType` on the last line is invalid, we should know the type of `sParam` and whether it has an attribute named `deviceType`. The type of `sParam` is the parameter type of the function value assigned to the `onDeviceStatusChange` property of the object `nserviceModule`. Because that property is passed as an argument of the `nservice.registerManageCallback` function in the NService API:

```
[NoInterfaceObject] interface NServiceManager {
  ...
  void registerManagerCallback(
    NServiceDeviceManagerCallback monitorCallback);
};
```

the type of `nserviceModule.onDeviceStatusChange` is the parameter type of the `registerManageCallback` function, which is `NServiceDeviceManagerCallback`:

```
[Callback=FunctionOnly, NoInterfaceObject]
  interface NServiceDeviceManagerCallback {
    void ondetected(ManagerEvent event);
  };
```

which is a function type from `ManagerEvent` to `void`. Thus, the type of `sParam` is `ManagerEvent`:

```
[NoInterfaceObject]
  interface ManagerEvent {
    readonly attribute unsigned short eventType;
    readonly attribute DOMString deviceName;
    readonly attribute DOMString uniqueID;
  };
```

which does not have an attribute named `deviceType`. As the example shows, writing correct JavaScript web applications using Web APIs with rich structures is a challenging task for JavaScript developers and $\mathsf{SAFE}_{\mathsf{WAPI}}$ can help the developers build safe software by detecting possible errors during development.

6.3 Wrong number of arguments to API function calls

We found this misuse pattern in 1 mobile application as shown in the following simplified excerpt from mb-6:

```
var NService_Client = {
  multicastMessage: function(msg) {
    if (typeof NService_Client.deviceGroup
        == 'undefined') {
      ...
    } else if
      (msg ===
       NService_Client.DEV_EVENT_JOINED_GROUP) {
      result =
        webapis.nservice.multicastMessage(
          NService_Client.DEV_EVENT_JOINED_GROUP);
    } else if
      (msg ===
       NService_Client.DEV_EVENT_LEFT_GROUP) {
      result =
        webapis.nservice.multicastMessage(
          NService_Client.DEV_EVENT_LEFT_GROUP)
```

$\mathsf{SAFE}_{\mathsf{WAPI}}$ detects two errors from the application as follows:

```
nservice_client.js:406:33~406:106: [WebAPIError]
  The number of arguments to
  webapis.nservice.multicastMessage is 1;
  provide arguments of size 2.
nservice_client.js:408:33~408:104: [WebAPIError]
  The number of arguments to
  webapis.nservice.multicastMessage is 1;
  provide arguments of size 2.
```

and reports that while the `multicastMessage` function expects two arguments the callsites provide only one argument. Indeed, the Web API specifies that the function takes two arguments:

```
[NoInterfaceObject] interface NServiceManager {
  ...
  unsigned short multicastMessage(
    DOMString groupID,
    DOMString message);
};
```

6.4 Missing error callback functions

We found this misuse pattern in 3 TV applications and 1 mobile application. The following simplified excerpt from mb-20 shows an example:

```
(function(global, webapis) {
...
  exports.joinTeam = function (team) {
    webapis.nservice.joinGroup(
      team,
      function (group) {
        global.bs.log(
          "Group members: " +
          JSON.stringify(group.getMembers()));
    });
```

While the NService API allows to omit the last argument to `joinGroup` as follows:

```
[NoInterfaceObject] interface NServiceManager {
  ...
  void joinGroup(
    DOMString groupName,
    NServiceGroupSuccessCallback onsuccess,
    optional ErrorCallback? onerror);
};
```

because it is an error callback function, SAFE$_{\text{WAPI}}$ warns JavaScript developers as follows:

```
bs.network.js:119:9~121:11: [WebAPIWarning]
  Call to webapis.nservice.joinGroup is missing an
  error callback function; provide an error
  callback function.
```

to make sure that they are aware of missing error callbacks to encourage them to develop defensive programming.

6.5 Unhandled API calls that may throw exceptions

This pattern is the majority of the detected misuses of Web APIs in the real-world web applications. We found this pattern in 2 TV applications and 8 mobile applications. The following simplified excerpt from mb-20 shows the pattern:

```
try {
  webapis.contact.getAddressBooks(
    function (addressbooks) {
      var contact1 = addressbooks[0].get('ID#1');
    },
    function (err) {
      console.log(
        err.name+
        ' error is occurred on getting address '+
        'books');
    });
} catch (ex) {
  console.log(
    ex.name+
    ' is thrown on getting all address books');
}
```

While the developer wrapped the code with the `try` statement to handle uncaught exceptions from the call of `getAddressBooks` on the second line, the `try` statement cannot handle uncaught exceptions from the call of `get` on the fourth line. Because the function `get` is called inside a success callback function passed to the function `getAddressBooks`, the developer should wrap the call of `get` with the `try` statement inside the callback function. This pattern happens very often which SAFE$_{\text{WAPI}}$ detects precisely as follows:

```
testContactManagerUpdate.js:4:42~4:53:
  [WebAPIWarning] Function webapis.contact.get may
  raise an exception; call the function inside the
  try statement.
```

6.6 Wrong types of arguments to API function calls

We did not find this misuse pattern from the collected web applications but we encountered the pattern occasionally with test cases. Consider the following code:

```
var appEventCB = {
  oninstalled: function(appinfo) {},
  onupdated: function(appinfo) {},
  onuninstalled: function(id) {} };
try {
  ...
  appEventCB = null;
  webapis.application.addAppInfoEventListener(
    appEventCB);
} catch (e) {}
```

The above code assigns the `null` value to `appEventCB` hoping that it unregisters the event callback. However, the Web API provides the following two functions for registering and unregistering event callbacks:

```
[NoInterfaceObject] interface ApplicationManager {
  long addAppInfoEventListener(
    ApplicationInformationEventCallback
                                   eventCallback)
    raises(WebAPIException);
  void removeAppInfoEventListener(long watchId)
    raises(WebAPIException);
}
```

Also, because the parameter of `addAppInfoEventListener` is not nullable, passing the `null` value as an argument is invalid as SAFE$_{\text{WAPI}}$ detects as follows:

```
appMgr.js:21:25~21:80: [WebAPIError]
  Argument #1 of the function
  webapis.application.addAppInfoEventListener is
  wrong; the expected type is
  ApplicationInformationEventCallback.
```

6.7 Accesses to absent attributes of dictionary objects

We did not find this misuse pattern from the collected web applications either but this pattern is another source of occasional mistakes found with test cases. While dictionary objects are very useful to represent associative arrays, developers should make sure that the JavaScript objects correctly implement corresponding dictionary types defined in Web APIs. Consider the following code:

```
var calendar =
  webapis.calendar.getDefaultCalendar("EVENT");
var calendarEvent1 = new webapis.CalendarEvent(
  { startDate: new webapis.TZDate(2012, 3, 4),
    duration: new webapis.TimeDuration(3, "DAYS"),
    summary: "HTML5 Seminar" });
var calendarEvent2 = new webapis.CalendarEvent(
    { name: 'Birthday party' });
```

Two constructor calls of `CalendarEvent` on the second and the sixth lines expect an input of type `CalendarEventInit`:

```
[Constructor(
   optional CalendarEventInit? eventInitDict),
 Constructor(
   DOMString stringRepresentation,
   CalendarTextFormat format)]
interface CalendarEvent : CalendarItem { ... }
```

the definition of which is shown in Figure 2. While the input object of the first constructor call is valid because all of its properties are members of `CalendarEventInit` inherited from the `CalendarItemInit` dictionary, the input object of the second constructor call is invalid because its property `name` is not a member of `CalendarEventInit`. Thus, SAFE$_{\text{WAPI}}$ reports the misuse as follows:

```
calendar.js:18:30~18:66: [WebAPIError]
  No matching constructors for CalendarEvent;
  possible constructors are:
    Constructor(
      [optional] CalendarEventInit? eventInitDict)
    Constructor(
      DOMString stringRepresentation,
      CalendarTextFormat format).
```

Unlike JavaScript developers who make mistakes in tracking information about Web APIs, SAFE$_{\text{WAPI}}$ correctly checks the properties of objects using the pre-analyzed database of Web APIs.

6.8 Threats to validity

Our results may not generalize for the following reasons:

- One of the main contributions of this paper is to propose a new mechanism to analyze complex interactions between JavaScript code and API function implementations rather than proposing a new analysis for JavaScript programs. Thus, the precision of our mechanism depends on the precision of the base JavaScript analyzer.

- Because the base JavaScript analyzer, SAFE, is in the abstract interpretation framework, our tool may report false positives.

- While our analysis models hidden execution flows between JavaScript code and API functions via callback function calls, we may miss other kinds of execution flows between them that API specifications do not explicitly describe. For example, API function calls may produce some side effects like manipulating DOM, which affects the behavior of JavaScript web applications. Because our target Web API functions manipulate platform and device functionalities, we believe that they may not produce side effects that change the semantics of JavaScript applications.

- Our analysis models API functions with the assumption that an API function may invoke at most one callback function, which implies that it does not consider semantic effects due to different invocation orders of multiple callback functions in an API function. Because most API functions invoke callback functions to communicate their execution results with JavaScript code, we believe that our assumption holds most of the cases.

7. RELATED WORK

Modern web applications execute on various platforms and devices using multiple APIs to access platform-specific information [6, 14, 24]. Any leakage of sensitive data from devices to untrusted third-party developers will affect all the players: API providers, web applications, and end users. However, most APIs are vulnerable to security attacks, and even carefully designed APIs [12, 1] with static restrictions and dynamic checks have been reported security problems [26, 21, 22]. As Guha *et al.* sated [16]:

> "Designing and implementing these APIs securely, and verifying that this has been done, is hence an important challenge."

To develop safe JavaScript programs, researchers have studied syntactic checks [13], type systems [27, 9, 17], static analyses [15, 18, 19], and hybrids of static and dynamic analyses [21, 11] for JavaScript. Unfortunately, the most widely used JSLint verifies only simple syntactic checks and coding guidelines, type systems are too restrictive to be acceptable by JavaScript developers in the wild, and the performance and precision of JavaScript analyses are not yet ready for being practical. One of the biggest problems with analyzing JavaScript applications is to understand the authors' intentions. Because JavaScript is an extremely dynamic language, programs alone do not reveal whether developers make mistakes or aggressively utilize dynamic features.

As complementary or replacing solutions, several variants of the JavaScript language try to communicate clearly with developers. CoffeeScript [10] provides syntactic sugars such as list comprehensions and pattern matching inspired by Ruby, Python, and Haskell to JavaScript, and it compiles into JavaScript. ActionScript [8] is a dialect of JavaScript with rich data types and type checking at both compile time and run time. TypeScript [23] is a typed superset of JavaScript that compiles into JavaScript. It supports classes, modules, and interfaces to improve static checking and verification of TypeScript programs.

Instead of some form of JavaScript variants, many vendors use separate interface definition languages particularly designed for specifying interfaces such as Web IDL [7]. Because most web applications are JavaScript programs, the Web IDL specification describes how specifications written in Web IDL correspond to JavaScript constructs. Thus, explicitly specifying the authors' intentions using Web IDL effectively bridges the gap between JavaScript web application developers and platform developers.

8. CONCLUSION

We present $\mathsf{SAFE}_{\mathsf{WAPI}}$, a practical static analysis tool to analyze JavaScript web applications to detect misuses of Web APIs. Even though the general problem of correctly analyzing JavaScript programs using closed-source APIs efficiently is a very difficult problem, we identified its subset problem that is practically useful and effective. Instead of trying to infer programmers' intention from extremely dynamic JavaScript programs, we utilize Web API specifications written in Web IDL to clearly communicate with them. To address explosive execution flows to analyze due to asynchronous semantics by events and callback functions, we adopted simplifying assumptions from development experiences with real-world web applications. We achieve reasonable analysis precision by connecting execution flows between JavaScript applications and platform functions via modeling platform-specific frameworks.

$\mathsf{SAFE}_{\mathsf{WAPI}}$ does not verify absence of Web API misuses in web applications but it effectively detects bugs from real-world applications. In the course of developing $\mathsf{SAFE}_{\mathsf{WAPI}}$, we even detected inconsistencies in the Web API specifications and found some real bugs in sample examples included in the specifications. We plan to increase the coverage of Web API misuse detection by defining more misuse patterns found in real-world applications and by modeling various JavaScript libraries and platform frameworks. We also plan to automatically validate Web API specifications for their consistencies and any misuses of them in sample examples.

9. ACKNOWLEDGMENTS

This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A01003235 and NRF-2008-0062609), Samsung Electronics, S-Core, and Google.

10. REFERENCES

[1] Caja. `http://code.google.com/p/google-caja`.
[2] ECMAScript Language Specification. Edition 5.1. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.
[3] HTML5. `http://www.w3.org/TR/html5/`.
[4] SAFE: Scalable Analysis Framework for ECMAScript. `http://safe.kaist.ac.kr`.
[5] Samsung Smart TV apps developer forum. `http://www.samsungdforum.com/`.
[6] Samsung web API on developer site. `http://developer.samsung.com/samsung-web-api`.
[7] Web IDL. `http://www.w3.org/TR/WebIDL`.
[8] ActionScript.org. ActionScript. `http://www.actionscript.org`.

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[10] J. Ashkenas. CoffeeScript. `http://coffeescript.org`.

[11] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[12] D. Crockford. ADsafe. `http://www.adsafe.org`.

[13] D. Crockford. JSLint. `http://www.jslint.com`.

[14] L. Foundation. Tizen. `http://tizen.org`.

[15] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th International Conference on World Wide Web*, 2009.

[16] A. Guha, B. Lerner, J. G. Politz, and S. Krishnamurthi. Web API verification: Results and challenges. In *Analysis of Security APIs*, 2012.

[17] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[19] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th International Symposium on Static Analysis*, 2010.

[20] 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 2012 International Workshop on Foundations of Object-Oriented Languages*, 2012.

[21] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *14th European Symposium on Research in Computer Security*, 2009.

[22] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, 2010.

[23] Microsoft. TypeScript. `http://www.typescriptlang.org`.

[24] Mozilla.org. Firefox OS. `http://www.mozilla.org/en-US/firefox/os/`.

[25] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the 7th IEEE International Symposium on Empirical Software Engineering*, 2013.

[26] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[27] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proceedings of the 14th European Symposium on Programming*, 2005.