DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF KAISERSLAUTERN

Master Thesis

# Information flow tracking for JavaScript in Chromium

Jonas Peschla

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF KAISERSLAUTERN

Master Thesis

# Information flow tracking for JavaScript in Chromium

|  |  |
|---|---|
| Author: | Jonas Peschla |
| 1st Supervisor: | Prof. Dr. Dieter Rombach |
| 2nd Supervisor: | Prof. Dr. Alexander Pretschner |
| Advisor: | Enrico Lovat, M.Sc. |
| Date: | December 14, 2012 |

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema "Information flow tracking for JavaScript in Chromium" selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 14. Dezember 2012 Jonas Peschla

# Abstract

Data usage control is a concept that extends access control to also protect data after it has been released. Usage control enforcement relies on available information about the distribution of data in the monitored system. In this thesis we introduce an information flow tracking approach for JavaScript in order to enable usage control for dynamic content in web browsers. The proposed model is implemented as a prototype in the JavaScript engine V8 of the Chromium browser.

# Acknowledgments

First of all, I thank Prof. Alexander Pretschner for supervising this thesis and encouraging me to work in the interesting field of usage control. Before this thesis I already worked together with his working group for about two years. I always enjoyed the great atmosphere and challenging topics we worked on.

Another word of thanks goes to Enrico Lovat, the probably only person on earth who knows everything, but from time to time pretends to not know the answer to a question in order to not discourage his fellow human beings. He advised me during this thesis and helped me a lot to concertize my ideas for the presented approach.

# Contents

# List of Tables

# List of Figures

# Listings

# 1 Introduction

This thesis deals with protection of third party data in web browsers. Today, websites often use JavaScript to provide a flexible user interface. This is achieved by asynchronous manipulations of the *Document Object Model* (DOM) of rendered pages [1]. Thereby, JavaScript may also push around data which is sensitive to its publisher.

We will introduce an approach to track flows of sensitive data through client-side JavaScript. The gained knowledge can be used to protect data from undesired usage. Our solution assists an existing usage control system for the Chromium browser which protects static data on rendered pages. As it operates on the DOM, it could detect modifications, but not whether and, if so, which sensitive data was involved in changes of the DOM. By inserting exactly this information gained through data flow tracking, we empower this system to also protect sensitive data in non-static settings.

The following examples illustrate the problem scope and motivation behind this work in a practical context.

## 1.1 The route planning example

Alice recently relocated. She published her new address in her social network profile to inform her friends about it. She only wants her friends to get to know her address, so she configured her profile accordingly.

Dave, an old friend of hers, decides to visit Alice. On her profile page he detects a route-planning feature which allows him to get a route to her address. The feature relies on an embedded third-party script, such as Google Maps. Depending on the profile information used to invoke the route-planning service, the feature could violate Alice's privacy. If, besides the bare address, also her name was be passed, this would basically violate what she specified in her profile.

### 1.1.1 Issues with third-party scripts

Websites allowing their users to configure privacy settings usually only apply them against other users, but not included services. This requires the user to make all-or-nothing decisions for information he considers to share. Even if he trusts the primary website provider, he still must decide this question for *all* embedded services *at once*.

And even if website providers cared about which data is sent along to third-parties, the way embedded scripts are currently handled in browsers makes it impossible to prevent leakage. In this thesis we introduce a possible solution for this problem.

## 1.2 The photo gallery example

In a second scenario, Alice uploaded pictures from her housewarming party. Again she limits access, this time to all attendees of the party.

Now Carol, who has been at the party, sits at home with her friend Bob who was not there. Together they view the pictures. When they come across a cute photo of Alice, Bob, who has a crush on her, asks Carol to copy this picture on his USB stick. As they are good friends, Carol does not mind and copies the picture. That this could be against Alice's will does not even come to her mind.

### 1.2.1 Issues with dynamic content

The previous example has already been used in the diploma thesis of Patrick Wenz [2]. There an assumption was, client-side JavaScript does not modify rendered content. He developed a solution to protect static HTML content in the Chromium browser which prevented Bob from copying Alice's picture.

For actual websites the assumption of only static content is not realistic. Thus, to put the proposed solution into practice, it must be enabled to also cover dynamic modifications.

Photo galleries on the web often provide thumbnail previews. When a thumbnail is clicked, the respective picture is displayed in large. Such features usually rely on JavaScript.

The information flow tracking approach introduced in this thesis can support the previous system to also protect dynamic content. When the DOM is modified, it adds usage control information tracked through JavaScript, which will then be interpreted by the existing system again to protect the changed parts.

## 1.3 Information flow

As indicated, current browsers cannot prevent data leakage through embedded scripts. Third-party scripts get the same privileges as site internal ones, and thus cannot be controlled by the embedding site. This is the reason why websites can not do any better than trust third parties, even if they cared about whether and which data is used by them.

To be able to give guarantees, it would be necessary to have browser-side mechanisms in place that detect attempts to leak sensitive data. Our proposed solution for this deficiencies is based on information flow tracking for JavaScript.

### 1.3.1 Kinds of information flow

In the context of information theory and programming languages, information flow characterizes the transfer of information from one variable into another. The literature distinguishes two kinds of information flow, *explicit* and *implicit*.

*Explicit information flow* occurs in assignments as in Listing 1.1 where the information associated with the data contained in a variable $y$ directly flows into a variable $x$.

```
x = y;
```

Listing 1.1: Example for explicit information flow

*Implicit information flow* is caused by control flow dependency or covert channels. We do not go into detail for covert channels, e.g. the duration of a computation.

For control flow dependency, the information associated with variables in branching conditions indirectly flows into the governed branches, no matter whether they are executed or not. Both variants are illustrated in Listing 1.2. *Implicit information flow I* denotes flows due to a taken branch. In the example the assignment of 1 to $l$ encodes implicit flow I from $h$ to $l$.

On the other hand, *implicit information flow II* characterizes possible observations due to a non-executed branch. In the given listing, after the if-then-else statement, $p$ would still hold 0. Based on that, one could infer the value of $h$ because $p$ was *not* changed.

```
h = true;
p = 0;
if (h == true){
  l=1;    // implicit information flow type 1
} else{
  l=0;
  p = 5;  // implicit information flow type 2
}
```

Listing 1.2: Example for implicit information flow

## 1.4 Related work on information flow analysis

Information flow analysis is often employed to investigate programs with respect to properties like confidentiality, integrity and non-interference. In the context of browser-side JavaScript, related work usually considers two kinds of variables, public and secret ones. Extensive work dealing with malicious third-party scripts exists in this area [3–6].

However, non-interference, as a static property concerning all possible runs of a program, has lost focus in this area. Pure static analysis for JavaScript rendered out to be impractical due to the high dynamism of the language which allows code-generation at runtime, e.g. instantiate new functions from strings or evaluate arbitrary code via the *eval* construct [3, 6]. It rejects all programs that create code from strings which are not known prior to execution, although they not necessarily violate confidentiality or integrity of secrets. Accounting for this, Magazinius, Russo and Sablefeld investigated how non-interference could be approximated by runtime mechanisms in a way that suffices practical requirements [5].

### 1.4.1 Runtime information flow tracking

Instead, research concentrated on runtime tracking mechanisms which observe and monitor information flows just as they occur. The goal here is to establish confidentiality and integrity for certain variables or locations in a website's context [3, 6]. Some approaches also combine static and dynamic methods to hybrid solutions [4, 6].

As the execution of JavaScript progresses, the mechanisms track whenever flows from secret into public variables occur and taint them too be also secret. When a statement would violate confidentiality or integrity of a governed secret value, execution is halted.

Besides this coarse grained classification there are also solutions that can distinguish different origins and keep a combined tainting for variables [4].

As said, considered work in this area shares the assumption of public and secret information on a website which is held in variables that are ordered in an according security lattice. Because we aim at supporting *usage control*, we follow a different approach. We do not distinguish sensitivity levels but data itself. We will come back to this in Section 1.5.

### 1.4.2 Data and representation

Another difference is that we distinguish between information associated with data and its actual representation. To us, information is the abstract contents that inheres in a particular data representation. We use the terms *information* and *data* interchangeably and explicitly state when we refer to representation. For the subject of related work this is not of importance as it only needs to classify data as secret or public. Furthermore, as data remains in the same level of abstraction, there is no need to consider data independently of its representation [7].

Regardless of this two differences we can leverage results from the introduced related work with respect to information flow tracking in general. The distinction lies in the reaction to and interpretation of tracked flows.

## 1.5 Usage control

Today, mature facilities to restrict data access exist [8]. But, the data *provider* usually looses control in the moment access is granted. Data *consumers* are free to further use it as they like, even if the provider disagrees.

Usage control generalizes access control such that also can be specified what consumers may or may not do with accessed data. Examples for such provisions are "*do not copy or print my picture*", "*delete this file after 10 days*" or "*do not watch this clip more than 3 times*".

On access, when the requester was verified to have proper usage control mechanisms in place, the applicable policy is bundled with the data and both are shipped over. Otherwise, access is simply denied.

### 1.5.1 Usage control enforcement

Usage control enforcement is based on the observation and possibly also modification of usage events. Whenever data is read, written, moved or transformed, this is some sort of data usage. Such usage events can be observed on different layers of abstraction. For example on the processor level, in applications like browsers or word processors or at the operating system level with file system, processes and virtual memory.

Literature distinguishes two kinds of policy enforcement. *Detective enforcement* observes occurring usage events and detects policy violations as they happen. Violations trigger compensating actions like informing the data provider about the non-compliant use. A real world example used in the literature is speed monitoring. Drivers cannot be hindered from driving too fast, but they are fined for violating the speed limit.

Unlike detective enforcement, *preventive enforcement* can also avoid policy violations. It requires actual data usage to be indicated by preceding requests. Requests are intercepted

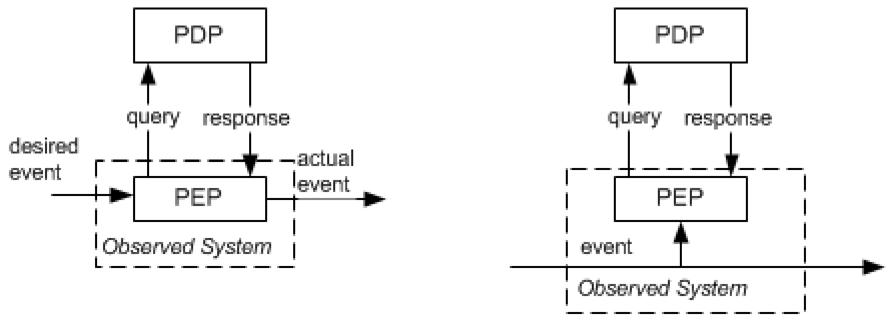


Figure 1.1: Conceptual view on preventive (left) and detective (right) usage control enforcement.

and checked against the active policies. If an intended usage violated a policy, the according request would either be inhibited or modified to become compliant.

The conceptual view, shown in Figure 1.1, is the same for both. The *Policy Enforcement Point* (PEP) is attached to the controlled system where it observes or intercepts usage events. It queries the *Policy Decision Point* (PDP) whether the observed events are compliant with the active policies. For preventive enforcement, the PEP suppresses actual events if the PDP signals to do so.

### 1.5.2 Enforcement along different levels of abstraction

In general, on lower levels of abstraction the possible coverage of observable events is higher. Unfortunately, it is hard to associate low level usage events to concrete, tangible actions on higher levels of abstraction - for example to combine all machine instructions that belong to saving a picture from a website on the hard disk. Also, on the operating system level one cannot reliably relate opened files to particular windows of an application.

Application specific PEPs are limited to events that occur within the scope of their monitored applications. But they have a fine-grained understanding of the semantics of observed events and can distinguish data more precisely. A PEP for a browser knows which data resides in which cache files, but cannot observe when other applications access them.

Hence, the idea of multi-layer enforcement emerged [7]. Usage control monitors on different levels of abstraction provide precise control over events in their respective domain. They communicate with each other and thus overcome their limited range of sight. For example, the web browser PEP could signal the operating system level monitor the contents of cache files. If one of them was a PDF with sensitive content, the operating system level monitor could deny access to viewers that are not usage control enabled.

#### Varying representations of data

A circumstance that has to be dealt with in multi-layer enforcement is the different manifestations of data among the different levels of abstraction. A picture can be the contents of a file, an <img> tag on a website, a set of pixels in a window, the content of a variable pointing to some memory region and so on. But the information associated with the

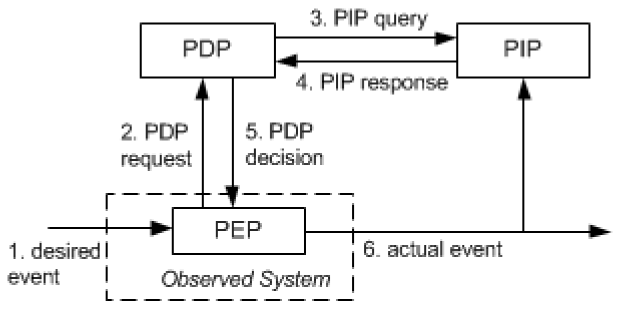


Figure 1.2: Extended usage control architecture for preventive enforcement. Policies are deployed in the PDP before the according data enters the observed system. This can be done by the data provider or the PEP.

picture is always the same.

When data flows from one layer of abstraction to another its representation usually changes, but not its meaning. This is the reason why we distinguish between *data* and *containers*. In the example, the file, the <img> tag and memory regions are different containers for the same data - the abstract information associated with the picture.

To realize cross-layer communication and system wide usage control, it is necessary to map corresponding events for usage actions on different layers of abstraction and identify which data flows from which container in the source layer to which container in the destination layer.

#### The Policy Information Point

Since the distinction between containers and data, the conceptual architecture for usage control systems has been extended by a third component, the *Policy Information Point* (PIP). It is the general data flow tracking component maintaining the associations between containers and the data present in them. Thus, it provides knowledge about the data distribution among the different layers of abstraction.

In the extended architecture, depicted in Figure 1.2, the PEP intercepts usage requests (1) and queries the PDP about them (2). To decide whether the actual usage would violate a policy, the PDP relies on information about the current data distribution from the PIP (3 and 4). Based on the received result (5), the PEP either grants or inhibits the usage (6).

## 1.6 Problem statement

Usage control enforcement in browsers has been investigated before [2, 9]. In both cases dynamic modification of rendered content through client-side JavaScript was excluded. However, a practical usage control system needs to also consider this aspect.

The problem tackled in this master thesis is the development of an information flow tracking model for JavaScript and its prototypical implementation. We aim at providing a basis to close the mentioned gap.

The developed model obeys the distinction between data and containers to account for multi-layer usage control. Furthermore, the prototype aligns with the general usage control architecture consisting of PDP, PEP and PIP.

### 1.6.1 Considered solution

For the information flow model, we will adopt and tailor the concepts already used for the same task on the operating system level [10] and the X11 window system [11].

A subset of statements and expressions from the EMCAScript language [12], which is the basis of JavaScript, will be formalized as transition system that evolves the state in the proposed model. However, the model is guided by V8's implementation of the standard.

### 1.6.2 Contribution

To the best of our knowledge, information flow tracking for JavaScript has not been investigated in the context of usage control so far. Related work focuses on data that is assigned to different sensitivity levels.

Thus, the main contribution of this thesis is to combine insights from general information flow tracking for JavaScript with the special needs of usage control in a tailored information flow tracking model. Data is distinguished by identity and tracked independently of its representation.

The feasibility of the proposed model will be evaluated by developing IF4JS, an instantiation of the model for the Chromium browser. It interacts with the DOM which constitutes a different layer of abstraction. IF4JS is a PEP, directly incorporated into Chromium's JavaScript engine V8 [13]. There it intercepts the code generation process to inject tracking instructions. Code rewriting is done by modifications of the intermediate *abstract syntax tree*[1] (AST) built by V8. Although we do not develop a full usage control enforcement system, IF4JS will be designed such that it could be easily integrated there.

## 1.7 Organization

The reminder of this thesis starts with an overview over the Chromium browser, the JavaScript engine V8 and the JavaScript language itself in Chapter 2. In Chapter 3 we define the information flow tracking model. Afterwards, the design of IF4JS is introduced in Chapter 4. The applied rewriting scheme is also covered there. Details about the implementation will be given in Chapter 5. In Chapter 6 we present an evaluation of the model and its implementation IF4JS. Finally, Chapter 7 will conclude the thesis and give directions for future work.

[1]**Abstract syntax trees** (short AST) are a concept to represent the syntactic structure of program code as a tree. The nesting of expressions and sub-expressions in statements is reflected in the sub-tree relationship of nodes. The semantics of particular child nodes depends on the parent's node type.

# 2 The Chromium browser

Chromium is an open source project actively developed by Google [14]. It serves as the basis for the Chrome browser which besides some additional features basically is "Chromium with Google's name and logo on it" [15].

## 2.1 Architecture overview

Chromium is not a browser from scratch. At its core it uses *WebKit* to render websites [16]. WebKit also is an open source project, developed under the lead of Apple. For Chromium the original JavaScript engine in WebKit, *JavaScriptCore*, is replaced by the highly optimized *V8* engine which will be covered in more detail below. On top of WebKit, Google implements the browser's multi-process architecture which uses separate processes for each displayed web page.

The details of Chromium's architecture in the layers above WebKit, which can be seen in Figure 2.1, are not of particular relevance for this thesis. As IF4JS will be incorporated into V8, it belongs to the WebKit layer and will not directly interact with the upper layers. For details about them please refer to Section 2.1 of Patrick Wenz's diploma thesis "Data Usage Control for ChromiumOS" [2] and the developer section on the project website [17].

How V8 is connected to WebKit, as well as the relevant parts of their architectures, will be covered in Chapter 4 where the design of IF4JS is presented.

## 2.2 The V8 JavaScript engine

V8 is Google's own JavaScript engine. Rather than being directly developed as part of the browser, it is a standalone project which provides an API to embed it. Hence, although initially and primarily developed for Chromium, it is not limited to this use case.

The rationale behind developing an own engine was the lacking performance of existing JavaScript engines at the time the development of Chromium started [18]. As websites made more and more use of JavaScript, in particular also Google's own services like GMail which is cited as example by the V8 team, its performance became vital for user experience.

V8 is known to be very fast. Three major reasons for its speed are the way it implements objects, properties and their modification; dynamic machine code generation and optimization during runtime; and its efficient garbage collector. Just as a side note, the second aspect has been investigated by Andy Wingo in detail [19].

### 2.2.1 The V8 API

To just execute a string of JavaScript code in V8 is simple. The example in Listing 2.1, which can be found in the "Getting started" section of the project page, shows how it works.

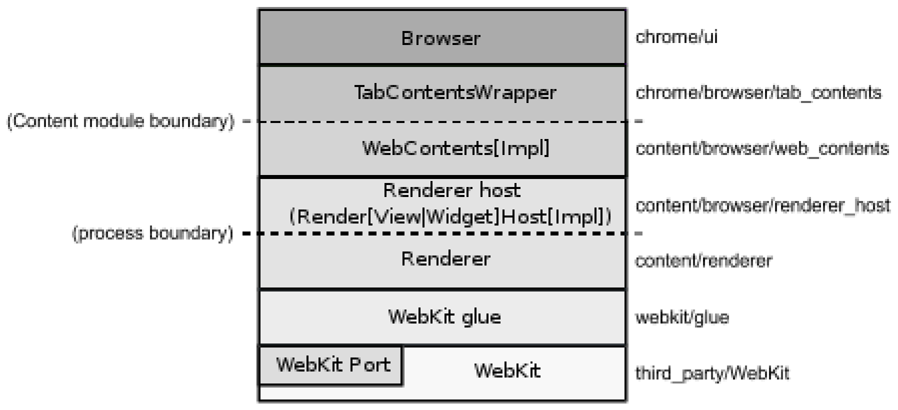


Figure 2.1: The conceptual application layers of Chromium. This diagram is a slight rework of an image taken from the Chromium project website and licensed under Creative Commons Attribution 2.5 license[1].

```
int main(int argc, char* argv[]){
  // Create a string containing the JavaScript source code.
  String source = String::New("'Hello' + ', World'");

  // Compile the source code.
  Script script = Script::Compile(source);

  // Run the script to get the result.
  Value result = script->Run();

  // Convert the result to an ASCII string and print it.
  String::AsciiValue ascii(result);
  printf("%s\n", *ascii);
  return 0;
}
```

Listing 2.1: JavaScript execution with V8

There are only three V8 classes involved: *Script*, *Value* and *String*. Their use is straightforward and one does not need to deal with any specifics of V8, not even *Contexts* [20]. But, without further preparation, executed code has very limited means. Why that is the case will be explained in 2.3, where we will shed light on JavaScript itself.

### Contexts

In V8, contexts serve the purpose to encapsulate a single runtime environment. Because JavaScript provides a set of built-in objects and functions which are mutable to executed

[1]Source: https://sites.google.com/a/chromium.org/dev/developers/design-documents/displaying-a-web-page-in-chrome

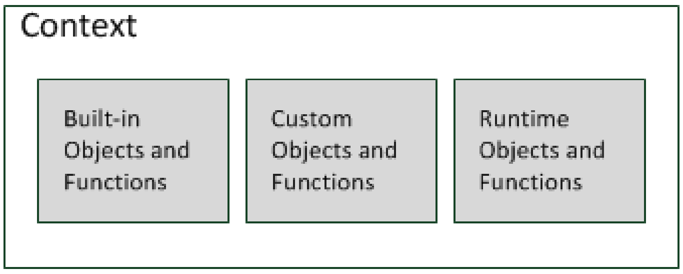

Context
Built-in Objects and Functions
Custom Objects and Functions
Runtime Objects and Functions

Figure 2.2: Illustration of a V8 context with built-in and external objects and functions. Runtime functions and objects are depicted separately because they are immutable.

code, it is required that each executed program runs in its own scope. Otherwise scripts from different web pages would likely interfere with each other.

Figure 2.2 shows an illustration of a context. The blocks "Built-in -" and "Runtime Objects and Functions" denote built-in capabilities and the objects executed code creates, respectively. "Custom Objects and Functions" indicates a third class of objects that can exist in a V8 context. The V8 API allows embedders to provide own objects and functions to a runtime environment and thus extend the capabilities for executed code.

### Function and Object templates

V8 provides the classes *FunctionTemplate* and *ObjectTemplate* to extend a context by additional objects and functions. Embedding applications may use them to reflect custom C++ code into JavaScript. *FunctionTemplates* allow to bind C++ functions as handlers that are invoked whenever JavaScript code calls the associated function names. *ObjectTemplates* let JavaScript objects be backed by C++ implementations. For them, V8 delegates property access to the respective C++ code.

In Chromium these mechanisms are used to introduce the DOM API to JavaScript.

### Handles and objects

The implementations of object and function templates need a way to access and modify values in the JavaScript runtime environment. V8 provides certain data types that are used to represent them internally. They are depicted in Figure 2.3. But, references to them are never exposed directly. The reason for this is how garbage collection works in V8. When the garbage collector conducts a collection cycle, it moves JavaScript objects around to optimize the memory layout. Thus, pointers to moved objects would become invalid.

To circumvent this problem, V8 introduces the concept of *Handles*. *Handles* are wrappers around the actual address of an object or value. The garbage collector knows about all existing handles and updates the location of the wrapped objects when necessary. Relocation remains transparent for the embedder as access to values is performed via *Handles*.

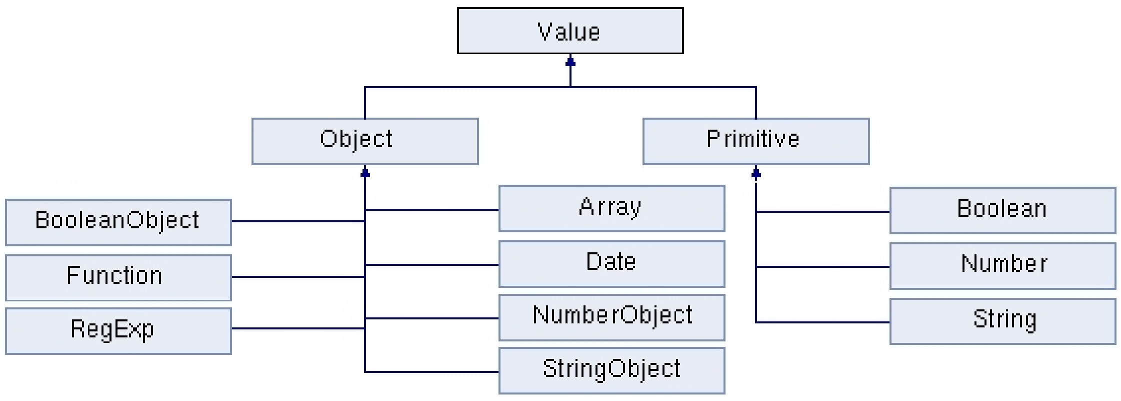


Figure 2.3: Type hierarchy of the built-in types in ECMAScript

## 2.3 JavaScript

JavaScript is understood as the scripting language enabling client-side modifications of rendered web pages. To enable modifications of rendered pages, browsers expose the underlying DOM to JavaScript.

Here lies a first differentiation that is important to be aware of: The DOM is technically not part of JavaScript as a language construct. A second circumstance is that JavaScript itself is a vague term. In the web area, JavaScript usually refers to the implementation of the ECMAScript language together with the DOM and the XMLHttpRequest API (XHR)[2] in a browser. From this perspective, JavaScript comprises all language constructs and APIs available to embedded scripts. Because implementations, especially of the DOM, differ between some browsers there are different dialects of JavaScript, so to say.

But JavaScript sometimes is also used as a synonym for the pure ECMAScript language. Throughout the remainder of this thesis, JavaScript is used to specifically refer to the implementation of ECMAScript in Chromium.

### 2.3.1 The formal basis: ECMAScript

ECMAScript is a dynamically typed, object-oriented, prototype-based language. Although originated from the standardizing process for scripting in browsers in the 1990s it is not limited to that. Extensions for OpenOffice.org can be written in an ECMAScript based language and Adobe specified an API that allows using it in PDFs. With Node.js[3] there even exists a framework that allows to write complete applications in this language.

ECMAScript has a limited set of predefined types and operations, because its focus lies on providing a uniform basis for scripting in general. But the implementing host environments are explicitly expected to add facilities that extend the capabilities of ECMAScript programs, as browsers do with the DOM. These so-called *host objects* can freely extend the functionality as long as they adhere to the specified behavior for ECMAScript objects.

[2]XmlHttpRequest is an API browsers expose to JavaScript in order to allow embedded scripts to asynchronously communicate with remote hosts.
[3]Node.js homepage: `http://nodejs.org/`

### 2.3.2 Core language characteristics

**Dynamic typing:** In ECMAScript the types of variables and expressions depend on the respective runtime values. This requires careful consideration in the rewriting process if the possible runtime types of a handled expression play a role.

**Object orientation:** Besides the values of the five primitive types *Undefined*, *Null*, *Boolean*, *Number* and *String*, each value in ECMAScript is an *Object*. For Boolean, Number and String there are also object types that wrap corresponding primitive values. Furthermore, functions are also first-class objects. They can be created, assigned to variables and passed as parameters during runtime. The difference to other objects is that they are callable.

**Object structure:** Objects are defined as collections of properties. Thus, they can be seen as a closure around a set of values. Properties define named associations between the owning object and some value. Some properties, such as *prototype* on function objects, have a special meaning within the specification.

**Prototyping:** ECMAScript, unlike class-based languages as Java, does not provide a static type hierarchy. Inheritance is realized via so-called prototypes. When objects are created, they get assigned a *prototype* which is either another object or *null*. As prototypes are objects themselves, they also have prototypes. Thus, each object has a prototype chain, which eventually is terminated by *null*.

If a property access to an object cannot be resolved locally, because the property name is not defined in that object, it is recursively looked up in the the prototype chain. As a consequence, objects sharing the same prototype also share all properties in their prototype chain which are not shadowed by local properties.

**Functions:** Because functions are first-class entities in the language, they behave different from methods in the sense of class-based languages. There, functions are bound to classes and called methods of the respective class. In ECMAScript, functions not necessarily belong to an object. But if they are bound as property, this affects the way they are invoked. In this case they are also called a *method* of that object.

**Function calls:** If a function is called, besides the actual values for the formal parameters, it also receives an additional parameter that serves as *ThisBinding*. The value associated with it will be used to resolve properties accessed via the *this* keyword in the function body. If a function is called as a property of an object, the runtime environment will automatically pass that object as *ThisBinding*.

**Functions called as constructor:** Functions can also be invoked as constructors. This is done by preceding the call by the *new* keyword. The returned result is a newly created object which is bound to the *ThisBinding* for the evaluation of the function body. Created objects get assigned a prototype which is determined by the special *prototype* property of the constructor function. It does *not* describe the function's own prototype. Figure 2.4 illustrates this and also shows an example for shared prototypes.

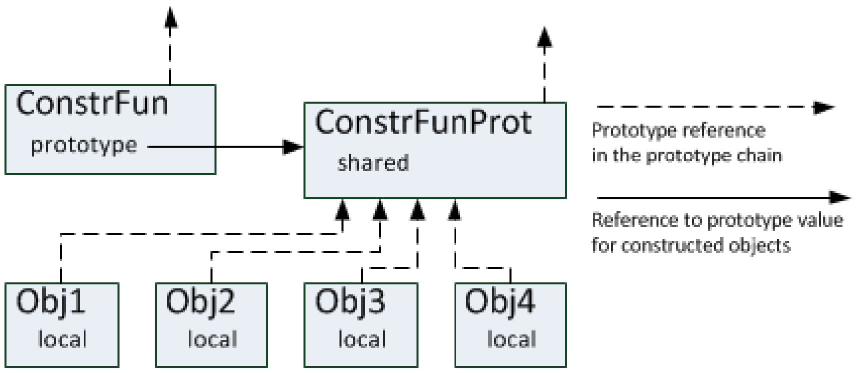

ConstrFun
prototype
ConstrFunProt
shared
Prototype reference in the prototype chain
Reference to prototype value for constructed objects
Obj1
local
Obj2
local
Obj3
local
Obj4
local

Figure 2.4: Illustration of the prototype mechanism: *ConstrFun* is a function which assigns *ConstrFunProt* as prototype to objects it creates when called as a constructor. This prototype reference is different from the function's own prototype as an ECMAScript object. This image is based on Figure 1 from the ECMAScript standard specification.

# 3 Information flow model for JavaScript

The information flow model for JavaScript is based on the principles introduced by Harvan and Pretschner [10]. Data and containers are used to model pieces of information and locations they are present in. Information flow is described by a transition system in which actions (statements and expressions) trigger modifications of the current state.

In JavaScript, containers are memory regions referred to by variables. Containers are either objects or primitive values like booleans, numbers or strings. A particular state is characterized by the distribution of usage controlled data among the containers. Because containers may be aliased, i.e. referred to by different variables, we must also model the mapping between variable names and the actual containers. Hence, the core entities in our model are the sets of containers $C$, data $D$ and container names $F$.

This sets are related to each other via three functions $f$, $s$ and $p$ that are used to describe the states $\Sigma$ in the model. The *naming function* $f$ maps variable names to containers, the *storage function* $s$ relates containers to sets of contained data and the *points-to function* $p$ keeps track of the nesting between objects and properties. Due to the way variables and properties are treated in JavaScript, the domain of the naming function is comparably complex. The details about this will be explained in 3.1.3.

## 3.1 Definitions

| | |
|---|---|
| $C$ | $= C_{Ref} \cup C_{Prim} \cup \{empty, undefined\}$ - set of all containers where:<br>$C_{Ref}$ denotes the set of all objects<br>$C_{Prim}$ is the set of containers of all primitive values<br>*empty*, *undefined* are reserved values, explained below |
| $D$ | set of usage controlled data items/ids |
| $B$ | set of lookup scopes for variable names |
| $V$ | set of all existing variable names |
| $P$ | set of all existing property names |
| $F$ | $= B \times (V \times (\bigcup_{n\geq 0} P^n))$ - set of container names where:<br>$B$ describes the scope in which a variable or property is looked up<br>$V$ identifies variable names to look up in a particular scope<br>$P$ is used to refer to properties within objects |
| $A$ | $= A_E \cup A_S$ - the set of supported actions (expressions and statements) |
| $s$ | identifier for the storage function $(C \to 2^D)$ |
| $s_p$ | identifier for the data-lookup function $(C \to 2^D)$ |
| $p$ | identifier for the points-to function $(C \to 2^C)$ |
| $f$ | identifier for the naming function $(F \to C)$ |
| $\Sigma$ | $= s \times p \times f$ - set of states |
| $R$ | $\subset (\Sigma \times A \times \Sigma)$ - the transition relation, see below |

### 3.1.1 Containers

Containers are conceptual entities in which data can be present. The presence of data is captured by the storage function $s$ that links a container to the identifiers of the data in it. In the model, the objects and primitive values in the JavaScript environment, respectively their memory locations, are considered to be the containers. We also define two special containers *empty* and *undefined* which are used as values for the naming function. When a variable holds no value but has been defined, it is mapped to *empty*. *Undefined* is the container for undefined variable names and helps to define the naming function as a total function.

**Objects and properties** For primitive values the relation between container and data is a simple 1:1 relation. This is different for containers that are associated with objects. Objects have properties that themselves either refer to primitive values or can point to other objects again. The model must be able to capture this kind of nesting.

**The points-to function** To this end we introduce the points-to function $p$. It describes the set of containers a certain container refers to. By this we can model the relationship between objects and their properties. For an object $o \in C_{Ref}$, $p(o)$ yields the set of containers for the properties of $o$. For containers in $C_{Prim}$ it always yields the empty set.

### 3.1.2 Data

Data comprises sensitive pieces of information that are subject to usage control. They are associated with an identifier, which allows to refer to them independently of their representation. The data flow model uses these identifiers in the storage function. Actions may perform updates on this function and by this capture data flow.

**The storage and data-lookup functions** The storage function $s$ describes which data is present in a container. More precise, it describes which data is directly associated with it. This distinction is important for objects because we consider them to also be containers for the data reachable through all their properties.

So, to determine the data present in a container, we define the data-lookup function $s_p$ which takes properties into account. When applied to some $c \in C$, $s_p(c)$ yields the set $s(c) \cup \bigcup_{\forall cp_i \in p(c)} s_p(cp_i)$. For primitive containers $prim \in C_{Prim}$, it holds that $s_p(prim) = s(prim)$, because they have no properties.

### 3.1.3 Variables and container identification

In order to track data flow in JavaScript we need to update the storage and points-to function. This requires to identify the containers affected by statements and expressions.

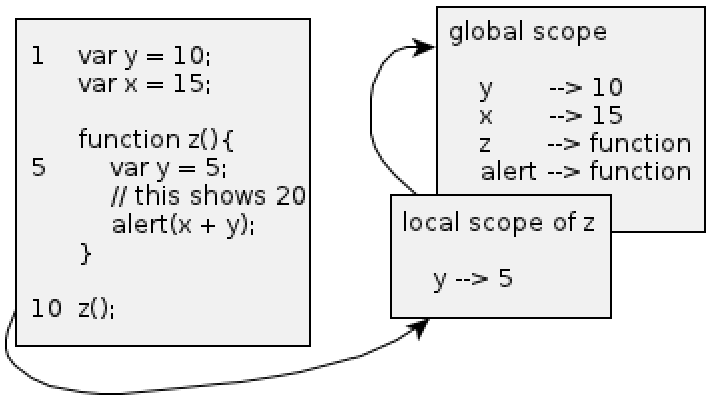


Figure 3.1: Illustration of the variable scoping in JavaScript. Throughout the execution of the lines 1 to 9 there is only the global scope. The call to $z$ creates a new local scope for the lifetime of the function body in which $y$ shadows the variable with the same name in the global scope.

**Variable handling in JavaScript** JavaScript uses hierarchical lookup scopes for variables. For a function call made in the global scope this means the body of the called function gets a new variable scope which is chained with the global scope as shown in Figure 3.1. The uppermost scope is the newly created one in which function local variables live. To look up a variable, first the topmost scope is inspected and then, if not found, the chain of scopes is consulted one after another until the variable name has been found. If the end of the chain is reached, the variable is resolved to *undefined*.

Consequently, we have to consider variable shadowing which means, a container can not be reliably identified by the name of a variable only. It is also important in which scope the variable name occurs. This explains why the set of execution scopes $B$ is part of the container identifiers $F$.

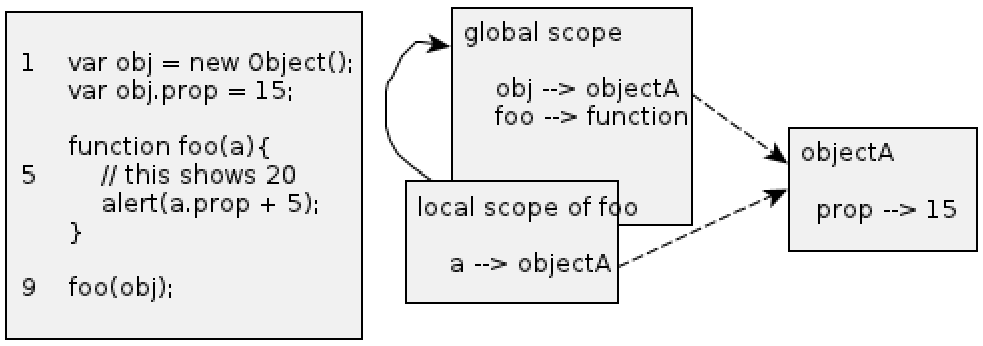
```
// container name would be (b, (someVar, ())), short (b, [someVar])
someVar;

// container name would be (b, (someVar, (property))),
// short (b, [someVar, property])
someObject.property;

// container name would be (b, (someVar, (property, subproperty))),
// short  (b, [someVar, property, subproperty])
someObject.property.subproperty;
```

Listing 3.1: Examples for how expressions map to naming function arguments in an assumed scope $b$

**The naming function** $f$ maps container names - which are tuples of a scope, a variable name and possibly a property name or property chain - to the associated container. The first component identifies the scope of a container name.



```
1   var obj = new Object();
    var obj.prop = 15;

    function foo(a){
5      // this shows 20
       alert(a.prop + 5);
    }

9   foo(obj);
```

Figure 3.2: In line 9, *foo* is called with the actual parameter *obj*. During its execution the formal parameter *a* thus points to the same object as *obj*. The solid line indicates the scope chain. The dashed lines represent object associations.

The second component is an n-tuple of type $V \times (\bigcup_{n \geq 0} P^n)$. This is in order to let $f$ resolve any syntactical references that are allowed in JavaScript - top-level variables, properties and sub properties. The elements of $V$ define variable names to be looked up. Property or sub property references are captured by an additional element of $P$ for each level of nesting. Listing 3.1 illustrates how JavaScript expressions in an assumed scope $b$ are mapped to container names. For the sake of brevity we will use $[var, prop_1, ..., prop_n]$ as syntactic sugar for $(var, (prop_1, ..., prop_n))$.

Note that calling the naming function $f$ with two different scopes $b$ and $b'$ and the same second component can yield the same container. In the scenario from Figure 3.1 this would be the case for $x$ in the local scope of z and the global scope. Furthermore, different variables or properties can point to the same object, as *obj* and *a* do in Figure 3.2. Hence, container names are not unique identifiers.

Using the lookup scope and a list of properties with arbitrary length, we can reliably distinguish containers and reflect any nesting depth.

## 3.2 Actions

Before, we explained how variables, containers and data are interrelated via naming, points-to and storage function ($f, p$ and $s$), which also describe the states in the model. We now formally define the actions and how they trigger information flow. The model evolution is described by a transition relation $R \subset (\Sigma \times A \times \Sigma)$ where $(\sigma, a, \sigma') \in R$ means that action $a$ causes the transition from state $\sigma$ to $\sigma'$. $R$ is the smallest relation that satisfies the equations 1 to 12 given below.

As $f, p$ and $s$ describe states, transitions are function updates. To describe a function update, we use the same notation as Harvan and Pretschner [10]. For $m : S \rightarrow T$ and a variable $x$ ranging over $X \subseteq T$:

$$m[x \leftarrow expr]_{x \in X} = m' \text{ with } m' : S \rightarrow T \text{ and}$$

$$m'(y) = \begin{cases} expr, & \text{if } y \in X \\ m(y), & \text{otherwise} \end{cases}$$

We will also make use of the semicolon as a shortcut to express simultaneous updates on disjoint sets:

$$m[x_1 \leftarrow expr_{x_1}; ...; x_n \leftarrow expr_{x_n}]_{x_1 \in X_1, ..., x_n \in X_n} = \\ m[x_n \leftarrow expr_{x_n}]_{x_n \in X_n} \circ ... \circ m[x_1 \leftarrow expr_{x_1}]_{x_1 \in X_1}$$

The set of actions $A$ reflects a subset of expressions ($A_E$) and statements ($A_S$) in ECMAScript. However, we developed the model based on the particular ECMAScript implementation in V8. We leave out statements that do not change the state or only do so via nested expressions and statements. Hence, we model none of the control flow statements. We assume implementations to cover such compositions, for example the branching conditions of If-statements.

We specify 11 actions: *Assignment, BinaryOperation, Call, CallNew, CallObject, Literal, Property, Variable, PropertyDeclaration, Return* and *VarDeclaration*. Actions that represent expressions - all but the latter three - yield a value which depends on the state they are executed in. Except for *Literal*, *Property* and *Variable* expressions, they also change the state. Implementations must make sure to not miss state changes in the observed system that are caused by nested expressions. IF4JS handles this by expression decomposition, which we will introduce in 4.2.4.

In our model each value represents a container and thus has a set of data associated. If a value flows into a variable or property by an assignment, or becomes part of a new value as for operands of binary operations, the set of associated data is mapped to the target. In JavaScript the statement type *ExpressionStatement* wraps expressions and simply evaluates them, which is often the case for assignments. In this case the resulting value has no effect. But the consequence is that model evolution can likewise be driven by expressions and statements.

### 3.2.1 Expressions

Whenever expressions occur in a statement or as sub expression, they are replaced with the result of their evaluation before the outer statement or expression is further processed.

***Literal*** and ***Variable*** are atomic expressions, i.e. they cannot have sub expressions which trigger intermediate computations. Literals yield a constant value which is never sensitive. Variables are constant names which are resolved within the current scope. To retrieve the data associated with a variable $var$ in scope $b$, $s_p \circ f$ is applied: $s_p(f(b, [var]))$.

***Properties*** are characterized by an object and a property name, which are given by expressions. In the simple case, the object is given by a variable. Otherwise, it is a nested property or a call expressions which both have to be evaluated first and then are replaced by their values before the actual property lookup is performed. The same holds for the name expression, which usually but not necessarily is given by a *Literal*. The result of a *Property* expression is the container associated with the property name in the identified object. Some examples can be found in Listing 3.2.

```
var obj = new Object();
// object given by variable, property 'foo' by string constant:
obj.foo;
// object evaluated by call, property by string constant:
getObject().foo;
// using a variable to access property 'foo':
var prop = "foo";
obj[prop];
```

Listing 3.2: Examples of different property lookup constructs

***BinaryOperations*** take two expressions, a left (*lop*) and right (*rop*) operand, and combine their results to a new value by applying a binary operator: *lop* ⊕ *rop*. Binary operations always produce primitive values which get associated the conjunction of the data linked to the involved sub expressions. *Lop* and *rop* can be any kind of expression. Nested binary operations are resolved recursively. For example, $var1 \oplus var2$ would be replaced by a new container $rv_{BOp}$ which is associated with $s_p(var1) \cup s_p(var2)$. In a more complex scenario as $(var1 \oplus getPrimitive()) \oplus obj.getPropObj().prop$, which can be found in Figure 3.3, first the left binary operation in brackets would be evaluated. For this, the value for *var1* is retrieved and combined with the result of the call to *getPrimitive* according to the operator semantics. The data associated with both is merged and bound to the temporarily created container. Then the outer binary operation could be evaluated, which requires to evaluate the right-hand side *Property* expression first. The method call to *getPropObj* on *obj* yields the object owning the property. Next, the container for the property *prop* is looked up. Finally, the remaining two containers are combined to the result value according to the outer operator.

As mentioned before, it may be the case that expression results are not used, what means they get no name. So, they in fact exist, but are not referenced and thus cannot influence further information flow. In JavaScript, the garbage collector deletes such values. However, operations that produce such values cannot be ignored as they may have side effects that must be captured.

### Assignments

Assignments cause explicit information flow. Their basic structure is *lhs* = *rhs*, meaning the information associated with the right-hand side expression flows into the left. Depending on the expression types of *lhs* and *rhs*, an assignment changes the current state in different ways. *Lhs* can either be a *Variable* or a *Property* expression which each could point to a primitive value or an object. *Rhs* can be any expression type, also yielding a primitive value or an object. When both sides hold primitive values, the assignment could be compound (+=, ...), so the operator itself also has an influence on the model evolution.

As assignments yield the assigned value, they are expressions instead of statements. This is relevant if an assignment occurs in a different context than an *ExpressionStatement*.

Based on the kinds of involved *lhs* and *rhs* expressions, we distinguish six kinds of assignments. Table 3.1 shows which combinations are handled by which assignment type. For the following specification, we assume assignments take place in scope *b*.

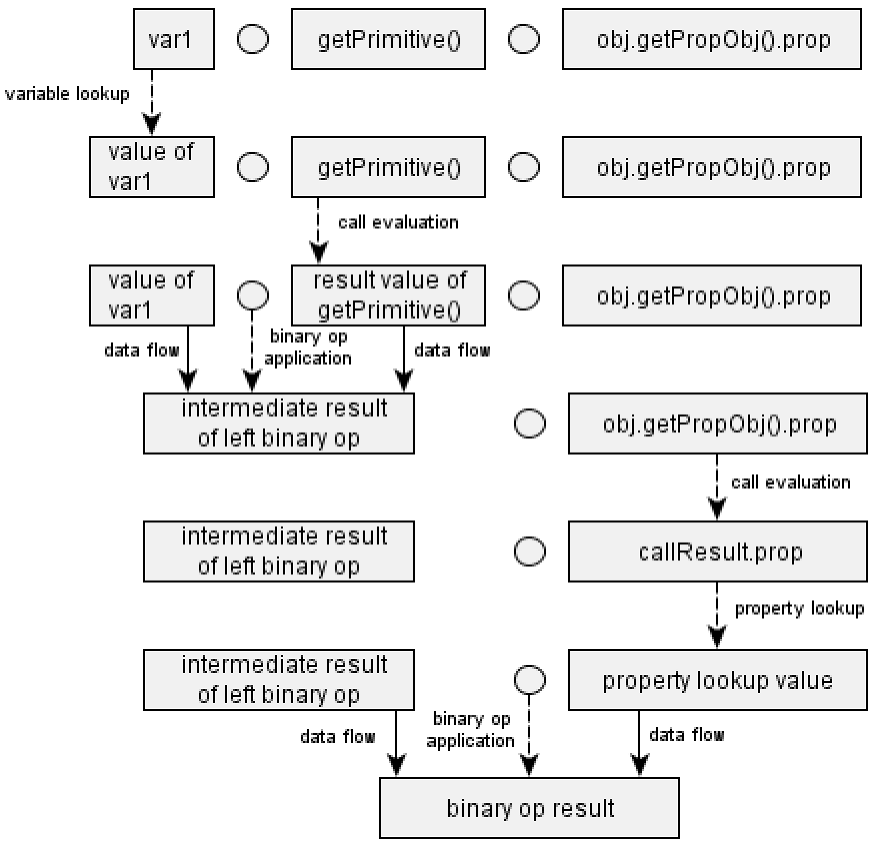


Figure 3.3: The steps for the evaluation of the binary operation $(var1 \oplus getPrimitive()) \oplus obj.getPropObj().prop$. Dashed lines describe the evaluation of a sub expression, solid lines indicate data flows.

| **Lhs expr.** | Variable | | | | Property | | | |
|---|---|---|---|---|---|---|---|---|
| **Lhs value** | Primitive | | Object | | Primitive | | Object | |
| **Rhs value** | Primit. | Object | Primit. | Object | Primit. | Object | Primit. | Object |
| **Assigment** | A1, CA1 | A2 | A1 | A2 | A3, CA2 | A4 | A3 | A4 |

Table 3.1: Overview of the six different assignment types in the model.

***Assignment1*** covers the two cases where a primitive value is assigned to a variable. Because simple values are copied on assignment, the *lhs* variable is associated with an unused container, i.e. one that had no name before, and gets assigned the *rhs* data:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \neg\exists\, n \in F : f(n) = c_{new} \wedge \\ &\quad ((s,p,f),(lhs = rhs),(s[c_{new} \leftarrow s_p(f(b,[rhs]))],p,f[(b,[lhs]) \leftarrow c_{new}])) \in R\end{aligned} \tag{1}$$

***CompoundAssignment1*** covers the case where a variable receives a compound assignment. Because the left value also flows back into the variable, the previously mapped data is merged with the *rhs* data and not overwritten:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \neg\exists\, n \in F : f(n) = c_{new} \wedge \\ &\quad ((s,p,f),(lhs\ op{=}\ rhs), \\ &\quad\ (s[c_{new} \leftarrow s_p(f(b,[rhs])) \cup s_p(f(b,[lhs]))],p,f[(b,[lhs]) \leftarrow c_{new}])) \in R\end{aligned} \tag{2}$$

***Assignment2*** handles both cases where a variable gets assigned an object. This changes the naming function as the variable name becomes a new alias for the object:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \\ &\quad ((s,p,f),(lhs = rhs),(s,p,f[(b,[lhs]) \leftarrow f(b,[rhs])])) \in R\end{aligned} \tag{3}$$

***Assignment3*** describes the assignment of a primitive value to some property *obj.prop*. The old property value is removed from the set of properties in the points-to function and replaced by a new container. The new container gets assigned the property name and the same data as *rhs*:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \neg\exists\, n \in F : f(n) = c_{new} \wedge \\ &\quad ((s,p,f),(obj.prop = rhs), \\ &\qquad (s[c_{new} \leftarrow s_p(f(b,[rhs]))], \\ &\qquad\ p[f(b,[obj]) \leftarrow (p(f(b,[obj])) \setminus \{f(b,[obj,prop])\}) \cup \{c_{new}\}], \\ &\qquad\ f[(b,[obj,prop]) \leftarrow c_{new}])) \in R\end{aligned} \tag{4}$$

***CompoundAssignment2*** describes the compound assignment of a primitive value to some property *obj.prop*. It is similar to Assignment3 but also maintains the data previously mapped to *lhs*:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \neg\exists\, n \in F : f(n) = c_{new} \wedge \\ &\quad ((s,p,f),(obj.prop\ op{=}\ rhs), \\ &\qquad (s[c_{new} \leftarrow s_p(f(b,[rhs])) \cup s_p(f(b,[obj,prop]))], \\ &\qquad\ p[f(b,[obj]) \leftarrow (p(f(b,[obj])) \setminus \{f(b,[obj,prop])\}) \cup \{c_{new}\}], \\ &\qquad\ f[(b,[obj,prop]) \leftarrow c_{new}])) \in R\end{aligned} \tag{5}$$

***Assignment4*** covers the assignment of a referential value to a property *obj.prop*. In the model, the link between the object and its old property value in the points-to function is replaced by a reference to the new value and the property name becomes an alias for the *rhs* object:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \\ &\quad ((s, p, f), (obj.prop = rhs), \\ &\qquad (s, p[f(b, [obj]) \leftarrow (p(f(b, [obj])) \setminus \{f(b, [obj, prop])\}) \cup \{f(b, [rhs])\}], \\ &\qquad\; f[(b, [obj, prop]) \leftarrow f(b, [rhs])])) \in R\end{aligned} \tag{6}$$

**Calls**

When a function is called via *Call*, *CallNew* or *CallObject*, a scope is added to the scope chain. In this scope the function body is evaluated. The call expressions describe the preparation of the scopes in which the formal parameters are bound to their actual values. The evaluation of function bodies happens according to the semantics of their statements and does not belong to call expressions themselves. The value returned by a call is determined by a corresponding *Return statement* which terminates the called function.

For each case, we assume a call to a function $fun$ with formal parameters $x_1, \ldots, x_n$. The calls are made from scope $b$ with actual parameters $a_1, \ldots, a_n$ and the function body will be evaluated scope $b'$.

***Call*** describes calls to functions that are not bound as method to a particular object. The expressions for the actual parameters are evaluated and their values bound to the corresponding formal names:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : ((s, p, f), fun(a_1, \ldots, a_n), \\ &\quad (s, p, f[(b', [x_1]) \leftarrow f(b, [a_1]); \ldots; (b', [x_n]) \leftarrow f(b, [a_n])])) \in R\end{aligned} \tag{7}$$

***CallObject*** handles functions called as a method of an object $obj$. Like before, the parameter binding is established and additionally $this$ is bound to the receiver:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : ((s, p, f), obj.fun(a_1, \ldots, a_n), (s, p, \\ &\quad f[(b', [x_1]) \leftarrow f(b, [a_1]); \ldots; (b', [x_n]) \leftarrow f(b, [a_n]); (b', [this]) \leftarrow f(b, [obj])])) \in R\end{aligned} \tag{8}$$

***CallNew*** covers constructor calls. For them, $this$ is bound to the newly created object which can be initialized in the body:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C] : \neg\exists\, n \in F : f(n) = c_{new} \;\wedge ((s, p, f), \\ &\quad new\; fun(a_1, \ldots, a_n), (s, p, \\ &\quad f[(b', [x_1]) \leftarrow f(b, [a_1]); \ldots; (b', [x_n]) \leftarrow f(b, [a_n]); (b', [this]) \leftarrow c_{new}])) \in R\end{aligned} \tag{9}$$

### 3.2.2 Statements

Statements are the basic building blocks for programs. Each program is a sequence of statements. Most statements trigger explicit flows by nested statements or expressions. In V8, this holds for all statements except for *Return*. Thus, it is the only V8 statement (compare Figure 4.4) we model directly. Source code constructs that declare variables or properties are transformed to special kinds of assignments and calls. Because it is easier for us to model them by separate actions, we additionally define *VarDeclaration* and *PropertyDeclaration* to capture them.

***VarDeclaration:*** When a variable $v$ is declared in scope $b$ (via *var v;*) this changes the naming function. A variable defined in a particular scope shadows variables with the same name in the scope chain.

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C]: \\ &\quad ((s, p, f), (var\ v), (s, p, f[(b, [v]) \leftarrow empty])) \in R\end{aligned} \tag{10}$$

***PropertyDeclaration:*** When a property $p$ is declared in an object $obj$ in scope $b$ this changes the naming function:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C]: \\ &\quad ((s, p, f), (declareProp(obj, p)), (s, p, f[(b, [obj, p]) \leftarrow empty])) \in R\end{aligned} \tag{11}$$

A property usually is declared in the course of an assignment to an undefined property name. If a property *prop* is nonexistent in *obj*, then *obj.prop = someVar;* first declares the property and then assign the value. The built-in object *Object* also allows property definitions via its *defineProperty*[1] method.

***Return:*** Return statements carry expressions which serve as results for corresponding function calls. Eventually, each function ends with a return statement. If there is no explicit one, *undefined* is returned. The return value replaces the corresponding call expression in the further evaluation of its context. To indicate this replacement we define $rv_{new}$ to be a new temporary name in the calling scope *b* which will refer to the result value. It is only valid while the expression context of the call is evaluated. The statement *return result;* in scope $b'$ triggers the following change:

$$\begin{aligned}&\forall s \in [C \to D^2], \forall p \in [C \to C^2], \forall f \in [F \to C]: \\ &\quad ((s, p, f), return\ result, (s, p, f[(b, [rv_{new}]) \leftarrow f(b', [result])])) \in R\end{aligned} \tag{12}$$

## 3.3 Application of the model

In the next two chapters we will introduce IF4JS, which is a prototype for information flow tracking in JavaScript based on the model above. As the language provides more statements and expressions than we specified in the model, we need mechanisms to cover unmodeled ones. Statement rewriting and expression decomposition will fulfill this purpose. How the runtime tracker maps actual JavaScript to actions in the model will be explained in 4.4.2. However, as a prototype it will not cover the whole language.

[1]Detailed description of *Object.defineProperty* on the Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/defineProperty.

# 4 IF4JS - Design

To evaluate the proposed model, we developed IF4JS as a PEP for JavaScript in Chromium. It is directly integrated into the JavaScript engine V8 and connected to WebKit in order to intercept DOM events in rendered pages. IF4JS tracks usage controlled data as soon as it is read from the DOM. When sensitive values are written to the DOM, IF4JS adds the tracked information from the model to the target. If sensitive data is about to be sent away using an XMLHttpRequest, IF4JS cancels the send operation.

## 4.1 Requirements

The general requirements for IF4JS are summarized in the following table:

| **ID** | **Description** |
|---|---|
| R-10 | IF4JS tracks JavaScript execution in Chromium using the model from Chapter 3. |
| R-20 | IF4JS reads UC information for sensitive data retrieved from the DOM. |
| R-30 | IF4JS adds UC information for sensitive data written to the DOM. |
| R-40 | IF4JS must follow the general UC architecture and consider PDP, PEP and PIP. |
| R-50 | IF4JS prevents sensitive data from being sent via XMLHttpRequest. |

Table 4.1: Requirements for IF4JS

### 4.1.1 Limitation of data sources

For the prototype we assume sensitive data only enters JavaScript via a certain set of DOM accesses which will be introduced below. As we want to extend the work done by Patrick Wenz [2], we also use the same mechanism to identify usage controlled data based on <span> tags. If the *class* attribute of a <span> tag matches a certain pattern, all child nodes are considered sensitive.

We are aware that the *XMLHttpRequest* API offers a second way, besides the DOM, to introduce sensitive data into JavaScript. Although we cover it as a sink for usage controlled data, we do not treat it as source here and leave this to future work.

### 4.1.2 Covered subset of the DOM API

Covering the whole DOM API would be unfeasible for the given time frame. Thus, for R-20 and R-30, we limit ourselves to a subset of its functionality for a proof-of-concept. We track read and write access to properties of arbitrary DOM nodes but do not track method invocations on them. The latter can not be handled by code rewriting because

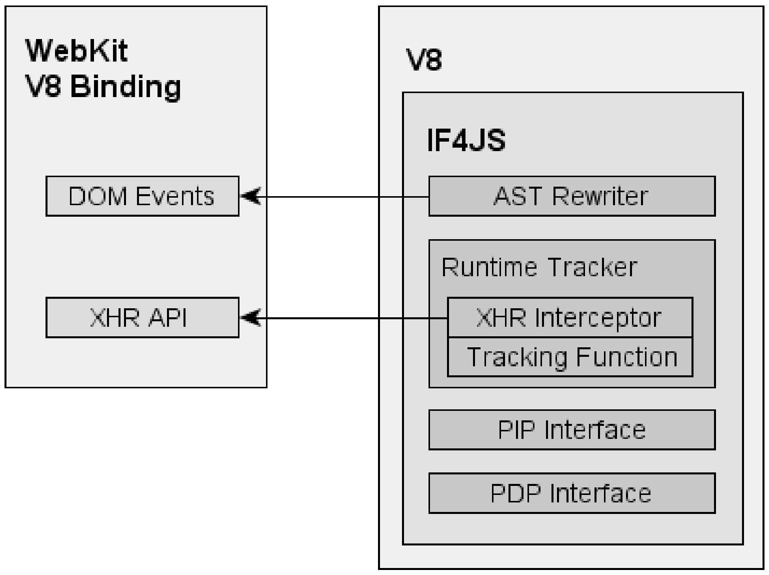


Figure 4.1: The conceptual architecture of IF4JS. The arrows describe control flow injections of IF4JS components into the V8 Binding.

the DOM has a native implementation. Tracking methods would require to modify their implementations analogously to the changes we made in the XHR API.

## 4.2 Design overview

The design of IF4JS is primarily influenced by three aspects:

1. Its integration in the interplay between WebKit and V8.
2. The alignment to the general UC architecture.
3. The chosen method to realize dynamic data flow tracking, which is code rewriting.

Figure 4.1 shows the conceptual architecture for IF4JS and how it is connected to WebKit. The concrete integration into V8 will be discussed later.

### 4.2.1 WebKit hooks

WebKit is the part in Chromium which renders web documents. It processes HTML, CSS and JavaScript. An overview of WebKit's layers can be found in Figure 4.2. In summary we made two modifications in WebKit. Both apply to the *Bindings* layer which is the interface through which V8 is accessed:

- We added a switch to activate information flow tracking when a DOM event occurs.
- We modified the XMLHttpRequest API to intercept the sending of AJAX requests.

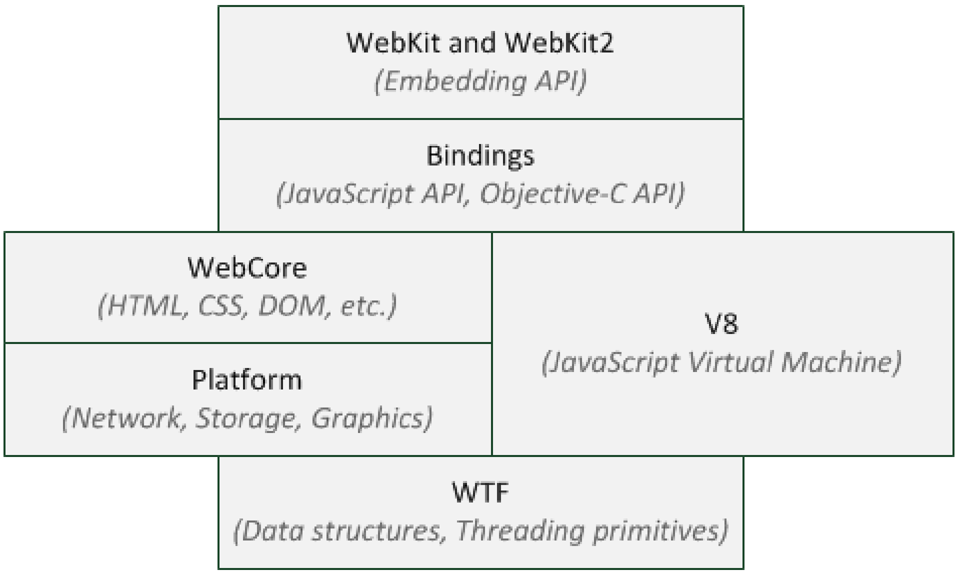

WebKit and WebKit2
(Embedding API)
Bindings
(JavaScript API, Objective-C API)
WebCore
(HTML, CSS, DOM, etc.)
Platform
(Network, Storage, Graphics)
V8
(JavaScript Virtual Machine)
WTF
(Data structures, Threading primitives)

Figure 4.2: The architectural layers of WebKit in Chromium. This figure is based on a slide of a presentation by Adam Barth [21].

**The information flow tracking switch:** *WebCore* loads and renders web pages, and thus controls JavaScript execution. But, JavaScript is also used to implement parts of the user interface, which we do not want to track. So we need a way to determine when *WebCore* executes scripts. There are two ways to execute JavaScript on a web page.

One is the immediate execution of JavaScript that is found during a page is parsed. This affects all code that does not define a function or an event handler. We detect this in V8 due to the way it is invoked. We refer to such code as *direct JavaScript* in the following.

For DOM events [22], as caused by user actions, this is not possible. Thus we located the spot in the V8 *Bindings* where events are passed into V8 and added instructions to activate IF4JS before and deactivate it after they are processed.

**Interception of XMLHttpRequests:** The XMLHttpRequest API is registered in V8 via the function- and object-template mechanism explained in Section 2.2. We added tracking code to the parts that are responsible for creating and sending requests. By this we can prevent leakage of sensitive data via requests.

### 4.2.2 Usage Control architecture

Following the general usage control architecture shown in Figure 1.2, IF4JS conceptually is a **PEP** for preventive enforcement. But for most cases it will only update the PIP to capture data flows and not query the PDP about intercepted actions. This is because in JavaScript data usage means data being used in statements or expressions. As the XHR API is the only unprotected sink for data, it is also the only part that requires PDP decisions. For the rest it is sufficient to track information flow and add usage control information when data is leaked to the DOM.

As we have no suitable PDP or PIP at hand, we implemented dummies for them with predefined interfaces. This allows to replace them by more elaborate ones when it comes to integrating IF4JS into a comprehensive usage control system without bigger modifications to the core PEP part later on.

As the **PDP** is only required for the XHR API and we have no existing server application prototype which would receive usage controlled data via AJAX messages, we stick to a hard coded policy that simply prohibits sending any sensitive data.

None of our former **PIP** implementations provides the points-to function, so we developed a basic solution for this purpose. It allows to maintain the data container mapping according to the model from Chapter 3.

### 4.2.3 PIP interface

The interface for the PIP provides six methods. They allow to add and remove containers, manipulate the mapping between containers and data, and of course the points-to function:

***AddData(container, data)*** reflects updates to the storage function $s$. It adds data to the mapping for a container. This method can be called with an empty second parameter to only introduce a new container.

***RemoveData(container, data)*** also updates $s$. It removes data from the mapping for a given container.

***GetDataInContainer(container)*** applies the data-lookup function $s_p$ to the given container.

***AddPointsTo(from, to)*** creates a link between two containers. This mirrors an update to the points-to function $p$.

***RemovePointsTo(from, to)*** is the inverse operation to *AddPointsTo*.

***RemoveContainer(container)*** is used as callback for the garbage collector (GC). The GC calls it to signal when it is about to delete an unreferenced container. As containers without a reference cannot be accessed anymore, we can safely remove them from the model.

A custom way maintain the naming function is not required. V8 itself implements it in order to perform its task and IF4JS uses built-in functionality to resolve container names.

### 4.2.4 Information flow tracking within in V8

Code given to V8 passes four stages: *Parsing*, *AST creation*, *Compilation* and *Execution*. Details about their sequence and implementation can be found at [23].

Our approach to integrate information flow tracking into JavaScript is a form of code rewriting. Code rewriting takes a given piece of code and transforms it. This transformation can add instructions, modify existing ones or remove them. IF4JS applies code rewriting to inject code that captures the effects of assignments and modifies existing statements

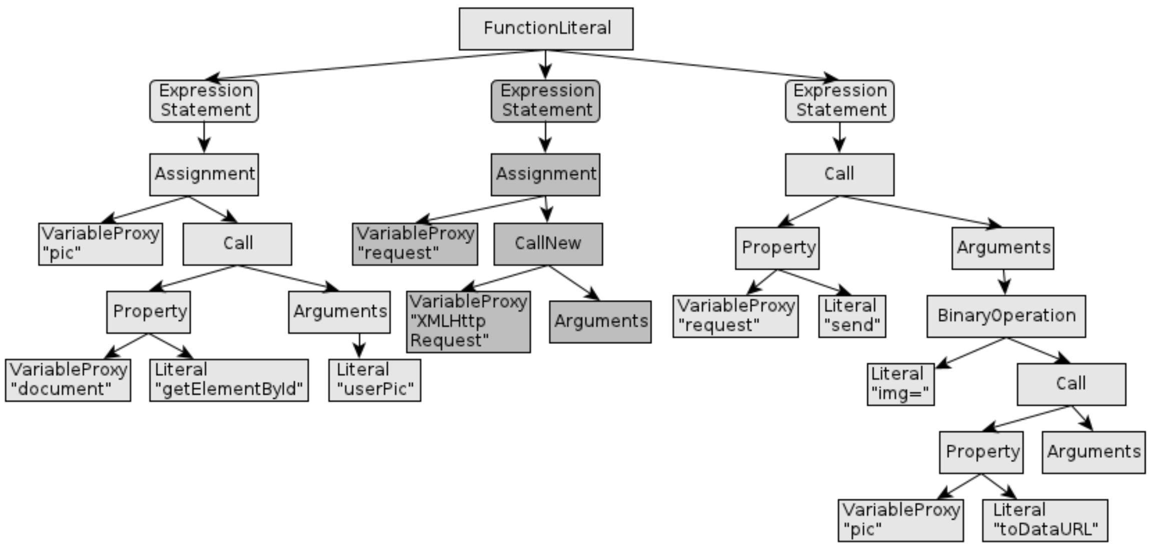


Figure 4.3: The abstract syntax tree V8 generates for Listing 4.1. Expressions are represented by rectangles, statements have rounded corners.

to be able to follow data flows step by step. This is achieved by decomposition of nested expressions into atomic steps.

## 4.3 Abstract syntax tree based rewriting

When WebKit renders a web page, it passes encountered blocks of continuous JavaScript to V8 for immediate execution. V8 parses these blocks and transforms them into *FunctionLiterals* which are root nodes of the ASTs used for compilation. If4JS intercepts these functions and rewrites the statements in their bodies to inject the runtime tracking system. We decided to use AST rewriting because it spares us the considerable effort of parsing the code on our own. Furthermore, it is easy to navigate through ASTs in V8. And for most statements and expressions, also modifications are simple. Encountered difficulties with switch statements and loops will be explained below.

An example AST can be found in Figure 4.3. It shows the AST V8 generates for the three lines of code in Listing 4.1. Besides the saved effort for parsing, ASTs provide typed nodes that allow us to easily map the represented statements and expressions to actions in our model.

```
pic = document.getElementById("userPic");
request = new XMLHttpRequest();
request.send("img=" + pic.toDataURL());
```

Listing 4.1: Example JavaScript code. Figure 4.3 shows the AST representation for it.

### 4.3.1 Coverage of AST nodes

There are two basic node types in V8, *Statements* and *Expressions*. The statements of a JavaScript program or function become siblings under the *FunctionLiteral* node representing the whole program or function in the corresponding AST. Depending on their concrete type, *Statements* contain *Expressions*, nested *Statements*, or a combination of both. In the following we introduce the existing AST nodes and explain which are covered by IF4JS' AST Rewriter to which extent.

**Statements:** V8 defines 16 concrete statement types, which can be found in Figure 4.4. Their structure differs depending on their type. *Blocks*, for instance, wrap a collection of nested statements. *IfStatements*, as another example, consist of a branching expression and two statements, then and else. *IfStatements* also illustrate one use of *Blocks*. They allow then- and else-branches to contain more than just one statement.

IF4JS fully rewrites *Blocks*, *Expression-*, *If-*, *Return-* and *TryFinallyStatements*. Although general rewriting rules for them exist, the rewriting of nested statements and expressions is of course limited to those types handled by the rewrite rules.

No rewriting rules exist for *Break-*, *Continue-*, *Debugger-*, *Empty-* and *WithStatement*. For *Break*, *Continue* and *Empty*, as they do not affect explicit flows, there is no need for rewriting anyways. *Debugger-* and *WithStatements* would have to be investigated by future work.

For loop statements, bodies are covered but not control expressions. In *Switch*, the expression used as switch and for *TryCatch* the variable in the catch construct are not rewritten. These limitations exist because handling them would require IF4JS to create new nodes of the respective types. Unlike for other statements and expressions, this requires a deeper understanding of the parsing and AST creation process than we gained so far. To close this gap, one most likely needs to find out how the required jump labels in these control flow statements are created and recalculate them.

**Expressions:** For eight out of the 19 expression types, see Figure 4.5, we defined actions in our information flow model and decomposition rules to rewrite them. These are: *Assignment*, *BinaryOperation*, *CompareOperation* (treated as *BinaryOperation*), *Call*, *CallNew*, *Literal*, *Property* and *VariableProxy* (as *Variable*). As described above, *FunctionLiterals* are also handled to integrate the runtime tracker. They are the starting point for the rewriting process.

Due to time constraints, we provide no mapping and rewrite rules for the remaining ten expressions. Covering them would require to identify the caused flows and either express them by already defined actions or introduce new ones. Deriving the rewriting rules requires careful consideration of the possible side effects of expressions in order to prevent semantic changes of instrumented programs.

Consequently, the covered expressions can only be fully handled by IF4JS if they do not use uncovered expression types as sub expressions.

### 4.3.2 Overview about the rewriting process

To rewrite an intercepted function, the *AST Rewriter* takes each statement from the function body, applies the rewriting rules and replaces the original statement by its rewritten version. Unhandled statements remain unchanged.

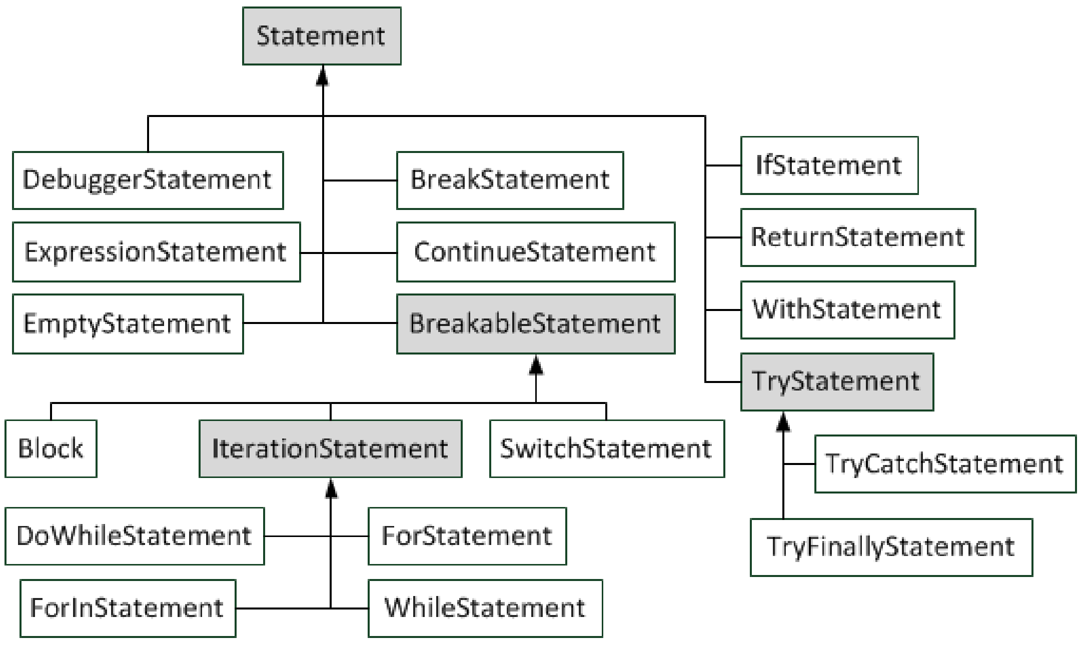


Figure 4.4: Overview about the statement types in V8.

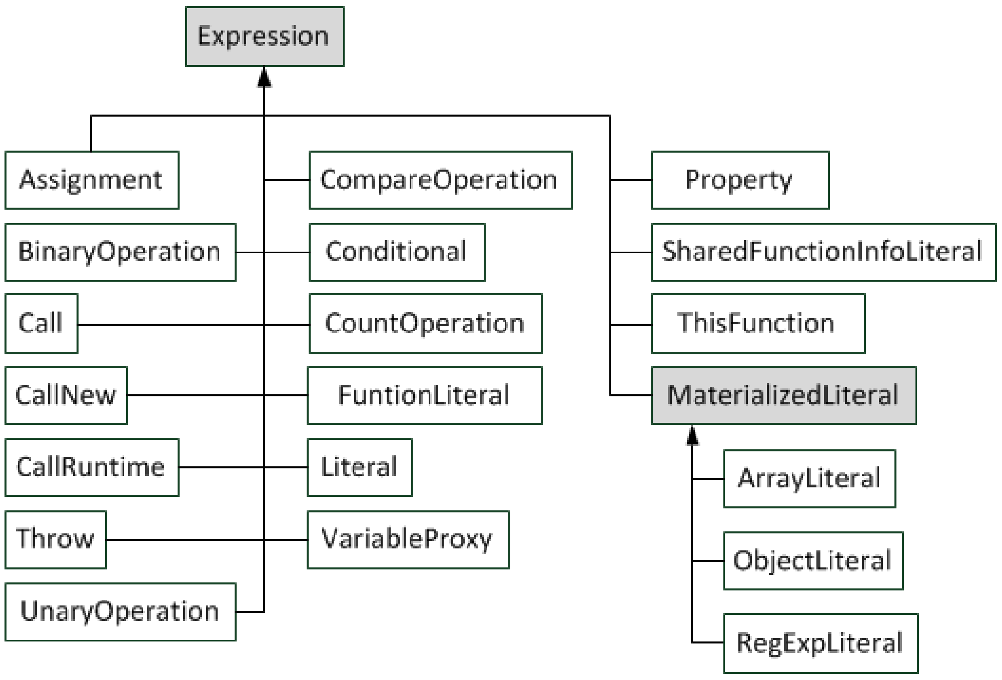


Figure 4.5: Overview about the expression types in V8.

We use a decomposition approach to track explicit flows in expressions. The rewriter takes expressions from handled statements and extracts nested sub expressions first. These are then prepared for isolated evaluation in newly inserted statements. The results are assigned to temporary variables and additional instructions trigger the runtime tracker to observe these single flows. Afterwards, the original statement is replaced by one with the same result but which uses the decomposed expressions. Listings 4.2 and 4.3 show the principle applied to a simple and a more complex example. Both describe how an *ExpressionStatement* assigning the result of a binary operation to a variable $v$ is rewritten.

```
// before rewriting:
v = a + b;
//rewritten code:
v = a + b;
if4js_trackStatement("ASSIGNMENT_VP_BOP", false, "v", "a", "b");
```

Listing 4.2: A simple example for code rewriting applied by IF4JS

In the first example, the *ExpressionStatement* wraps an *Assignment* expression which has a *VariableProxy* as target and a *BinaryOperation* as assigned value. The left and right operands of the binary operation again are *VariableProxies*. Looking up variables causes no intermediate data flows, so the rewriter simply inserts a tracking call after the given statement. The parameters of the call instruct the runtime tracker to update the storage function for $v$. The new value will be the union of the data sets currently associated with $a$ and $b$.

```
 1 // before rewriting:
   v += getObj()[propA] + objB[propB] + c;
 3 //rewritten code:
   // first track the left operand (getObj()[propA] + objB[propB])
 5   // 1) track the retrieval of the property owner
   tmp1 = getObj();
 7   // 2) track the property value
   tmp2 = tmp1[propA];
 9 if4js_trackStatement("ASSIGNMENT_VP_PROP",false, "tmp2",tmp1,propA);
     // 3) track right operand's value
11 tmp3 = objB[propB];
   if4js_trackStatement("ASSIGNMENT_VP_PROP", false, "tmp3",objB, propB);
13   // 4) track the nested binary operation
   tmp4 = tmp2 + tmp3;
15 if4js_trackStatement("ASSIGNMENT_VP_BOP", false,"tmp4","tmp2","tmp3");
   // next track the outer binary operation
17 if4js_trackStatement("ASSIGNMENT_VP_PRE", "v");
     // replacement for the original statement:
19 v += tmp4 + c;
   if4js_trackStatement("ASSIGNMENT_VP_BOP", true, "v", "tmp4", "c");
```

Listing 4.3: A more complex example for code rewriting applied by IF4JS

The second example also shows an *ExpressionStatement* with an *Assignment* to $v$. The first difference is that the assignment here is a compound assignment, which means data present in $v$ will not be overwritten but merged with the right-hand side. The call inserted in line 17 accounts for that. It triggers the runtime tracker to backup the data associated with $v$ in the current state before the compound assignment is performed. Afterwards, in line 20, the tracker will associate *v* not only with the union of the data for *tmp4* and *c*, but also add the previously saved data. Whether or not backed up data has to be considered is indicated by the second parameter of the respective tracking calls.

The compound assignment also has a *BinaryOperation* as right value, but in this example its left operand is a nested binary operation which causes intermediary data flows. Lines 6 to 15 show how the nested operation is recursively decomposed. The left operand (*getObj()[propA]*) of the nested operation is a *Property* expression. The owner object of the property is given by a *Call* expression which must be inspected for caused flows first. Thus, the property is rewritten such that the evaluation of the call can be tracked separately. The result of the call, which is the owner of the property, is stored in a temporary variable. This variable is used to rewrite the property lookup in line 8. In contrast to the original lookup it does not contain sub expressions that cause additional flows. Line 11 stores the second operand of the nested binary operation in a temporary variable and lines 14 and 15 finally capture the complete flow induced by the nested operation before the outer binary operation is tracked as explained above.

### 4.3.3 Rewrite rules

We use two functions, *instrument* and *decompose*, to describe the rewriting process. *Instrument* takes a statement and outputs instrumented statements for its replacement. It has no return value. *Decompose* takes an expression and returns a variable pointing to the decomposed result value. It also generates statements if required for the decomposition. How the outputs of both functions are used, respectively how they are stored, will be explained in the implementation part in the next chapter. We give the rewrite rules by pseudo code. Listings 4.4 to 4.11 show the rules applied for statements. The decomposition rules for expressions can be found in Listings 4.12 to 4.19.

***Block:***
*Blocks* are rewritten by rewriting all nested statements:

```
instrument(Block(stmt_1; ...; stmt_n;))

// is rewritten to:
Block(instrument(stmt_1); ...; instrument(stmt_n);)
```

Listing 4.4: The rewriting scheme for *Blocks*.

***ExpressionStatement:***
*ExpressionStatements* evaluate a wrapped expression. Thus, the rewriting process decomposes the expression:

```
instrument(ExprStmt(Expression))

// is rewritten to:
ExprStmt(decompose(Expression))
```

Listing 4.5: The rewriting scheme for *ExpressionStatements*.

***IfStatement:***
To rewrite *IfStatements*, the contained branching condition is decomposed and the then- and else-statement are instrumented:

```
instrument(IfStmt(Cond, Then, Else))

// is rewritten to:
IfStmt(decompose(Cond), instrument(Then), instrument(Else))
```

Listing 4.6: The rewriting scheme for *IfStatements*.

***ReturnStatement:***
*ReturnStatements* carry an *Expression* which yields the value passed back by a function call. The rewriter decomposes the return value:

```
instrument(Return(Expression))

// is rewritten to:
Return(decompose(Expression))
```

Listing 4.7: The rewriting scheme for *ReturnStatements*.

***TryFinallyStatement:***
This statement consist of two *Blocks* that contain the statements for the *try* and *finally* parts. The rewriter applies *instrument* to both blocks:

```
instrument(TryFinallyStmt(Try, Finally))

// is rewritten to:
TryFinallyStmt(instrument(Try), instrument(Finally))
```

Listing 4.8: The rewriting scheme for *TryFinallyStatements*.

***TryCatchStatement:***
Similar to *TryFinally* this statement has two *Blocks* for the *try* and *catch* parts, but there is an additional variable expression that is available in the catch block. The rewriter applies *instrument* to both blocks:

```
instrument(TryCatchStmt(Try, CatchVar, Catch))

// is rewritten to:
TryCatchStmt(instrument(Try), CatchVar, instrument(Catch))
```

Listing 4.9: The rewriting scheme for *TryCatchStatements*.

***IterationStatement:***
This common supertype of loops exposes the body of a particular loop. We use it to describe the rewriting for all loop types which only covers the body statement:

```
instrument(IterationStatement(Body))

// is rewritten to:
IterationStatement(instrument(Body))
```

Listing 4.10: The rewriting scheme for *loops*.

***SwitchStatement:***
*SwitchStatements* are composed from an expression that is switched upon and a set of case clauses. Cases consist of an expression that serves as their label and a block describing the statements of that clause. A possibly present *default* clause has no label expression. The AST rewriter rewrites the blocks of all case clauses:

```
instrument(SwitchStmt(Expr, Clause(L1,Block1),..., Clause(Ln,Blockn)))

// is rewritten to:
SwitchStmt(Expr,  Clause(L1,instrument(Block1)), ...,
                             Clause(Ln,instrument(Blockn)))
```

Listing 4.11: The rewriting scheme for *SwitchStatements*.

***VariableProxy:***
Variable expressions are not further decomposed. Thus, the *decompose* function simply returns the given expression and outputs no additional statements for them. We use the notation *VP(varName)* to describe a *VariableProxy* expression that points to the variable with name *varName* in the following decomposition rules.

***Literal:***
*Literals* represent constants. Their values are never sensitive. To ease the implementation the rewriter stores them in temporary variables and returns the according *VariableProxy*

```
decompose(Literal)

// outputs statements:
tmp = Literal;
//returns:
VP(tmp)
```

Listing 4.12: The decomposition scheme for *Literals*.

***Property:***
*Property* expressions consist of two nested expressions that identify the owner object and the name of the property. To decompose a property, the object and name expressions are recursively resolved (lines 4 and 5). The resolved values are used to perform the original property lookup and the lookup result is assigned to a temporary variable (line 6). The temporary variable is returned, as it holds the original expression's result:

```
   decompose(Property(objExpr, nameExpr))
 2
   // outputs statements:
 4 tmp1 = decompose(objExpr);
   tmp2 = decompose(nameExpr);
 6 tmp3 = tmp1[tmp2];
   if4js_trackStatement("ASSIGN_VP_PROP", false, "tmp3", "tmp1", "tmp2");
 8
   //returns:
10 VP(tmp3)
```

Listing 4.13: The decomposition scheme for *Properties*.

***Call and CallNew:***
*Call* and *CallNew* expressions consist of an expression that identifies the called function and a list of parameter expressions. IF4JS decomposes the parameters of calls and replaces them with calls using the decomposed values as parameters. The call result is stored in a temporary variable which is also the return value for the decomposition:

```
decompose(Call(funExpr, [p_1, ..., p_n]))

// outputs statements:
tmp_1 = decompose(p_1);
...
tmp_n = decompose(p_n);
tmp_r = Call(funExpr, [tmp_1, ..., tmp_n]);
//returns:
VP(tmp_r)
```

Listing 4.14: The decomposition scheme for *Calls*. The same scheme is applied to *CallNew*.

***Binary- and CompareOperation:***
These two expression types contain two nested expressions that provide the left and right operands of the operation. The operation is given by a binary operator. Rewriting is performed by decomposing both operands and replace the original operation by one that uses the decomposed values as operands. The applied scheme is the same for both:

```
decompose(BinOp(leftExpr, op, rightExpr))

// outputs statements:
tmp1 = decompose(leftExpr);
tmp2 = decompose(rightExpr);
tmp3 = BinOp(tmp1, op, tmp2);
if4js_trackStatement("ASSIGN_VP_BOP", false, "tmp3", "tmp1", "tmp2");
//returns:
VP(tmp)
```

Listing 4.15: The decomposition scheme for *Binary-* and *CompareOperation*.

***Assignment:***
*Assignments* also have two sub expressions. The target expression, which is either a *VariableProxy* or a *Property*, determines the left hand side. The value expression describes the right-hand side, the assigned value and can be any expression type. The rewriting applied for *Assignments* depends on the target expression type and whether the assignment has a simple or a compound assignment operator. Thus, there are four different rewrite rules:

```
decompose(Assign(VP(target), valueExpr))
// example:
target = value;

// outputs statements:
tmp = decompose(valueExpr);
target = tmp;
if4js_trackStatement("ASSIGN_VP_VP", false, "target", "tmp");

```

```
//returns:
VP(tmp)
```

Listing 4.16: The decomposition scheme for *Assignments* with a *VariableProxy* as target and a simple assignment operator.

```
decompose(Assign(VP(target), valueExpr))
//example:
target op= value;

// outputs statements:
tmp = decompose(valueExpr);
if4js_trackStatement("ASSIGN_VP_PRE", "target");
target op= tmp;
if4js_trackStatement("ASSIGN_VP_VP", true, "target", "tmp");

//returns:
VP(tmp)
```

Listing 4.17: The decomposition scheme for *Assignments* with a *VariableProxy* as target and a compound assignment operator.

```
decompose(Assign(Property(objExpr, nameExpr), valueExpr))
// example:
obj.name = value;

// outputs statements:
tmp1 = decompose(objExpr);
tmp2 = decompose(nameExpr);
tmp3 = decompose(valueExpr);
if4js_trackStatement("ASSIGN_PROP_PRE", false, tmp1, tmp2);
tmp1[tmp2] = tmp3;
if4js_trackStatement("ASSIGN_PROP_VP",false, tmp1, tmp2, tmp3);

//returns:
Property(tmp1,tmp2)
```

Listing 4.18: The decomposition scheme for *Assignments* with a *Property* as target and a simple assignment operator.

```
decompose(Assign(Property(objExpr, nameExpr), valueExpr))
//example:
obj.name op= value;

// outputs statements:
tmp1 = decompose(objExpr);
tmp2 = decompose(nameExpr);
tmp3 = decompose(valueExpr);
if4js_trackStatement("ASSIGN_PROP_PRE", true, tmp1, tmp2);
tmp1[tmp2] op= tmp3;
if4js_trackStatement("ASSIGN_PROP_VP",true, tmp1, tmp2, tmp3);

```

```
//returns:
Property(tmp1,tmp2)
```

Listing 4.19: The decomposition scheme for *Assignments* with a *Property* as target and a compound assignment operator.

***VarDeclarations*** have no direct counterpart in V8 and are realized differently for local and global variables. For variables declared without initial values, IF4JS does nothing as there is no explicit flow.

Local variables are declared via assignments which carry an initialization hint. IF4JS treats them the same way as other assignments. Global variables are declared through a call to the runtime function *InitializeVarGlobal* which happens through a *CallRuntime* expression. The applied rewriting, see Listing 4.20, uses the aforementioned expression decomposition to track the initial value.

```
CallRuntime("InitializeVarGlobal", variable, initValue);

// is rewritten to:
instrument(var tmp = initValue;)
CallRuntime("InitializeVarGlobal", variable, tmp);
if4js_trackStatement("ASSIGN_VP_VP", false, "variable", "tmp");
```

Listing 4.20: The basic rewriting scheme for the initialization of global variables. The suffix *_name* indicates that the name of a variable is used instead of its value

## 4.4 Dynamic information flow tracking

When V8 executes code compiled from modified ASTs, the Runtime Tracker comes into play. It consists of two parts. One is the tracking function *if4js_trackStatement*, which updates an instance of the proposed model maintained in parallel to JavaScript execution. The other part is the XHR API Interceptor, which essentially does the same for XHR instances.

### 4.4.1 The XMLHttpRequest Interceptor

We directly integrated data flow tracking for XHR requests into the API implementation, because this is less complicated than let flows be handled via AST rewriting and the tracking function. Additionally, it produces less runtime overhead.

The instructions to create and send requests are represented as calls in the AST. Whether a *CallNew* or *Call* expression will create or send a XHR cannot be reliably determined at compile time. The rewriter would have to inject tracking calls which for each call check, whether an XHR API call is about to be performed or not. These additional calls are unnecessary when data flow tracking is performed by the XHR API itself.

Furthermore, following this approach, the XHR interceptor can easily be extended to later on also handle PDP decisions that allow sensitive data to be sent away, but require a bundled usage control policy. From the perspective of the runtime tracking function, the methods of the XHR API are atomic actions. Because they are implemented natively, the AST Rewriter cannot inject tracking calls into their statements. Any actions performed by them would have to be anticipated and judged in advance. The interceptor has a more fine-grained view as it is interweaved into the implementation. It is tracking code injected by hand, so to say.

### 4.4.2 The runtime tracking function

The tracking function records information flow by observing assignments. IF4JS rewrites covered statements and expressions such that induced explicit flows are decomposed and pass through observable assignments. This makes it easy to track them. Basically, the tracking function looks up which data currently is associated with the right-hand value and updates the mapping for the target. Depending on the operand and operator types, the concrete steps differ in some details. For example, assignments to properties need additional steps to adjust the points-to relation. The six assignment types described in the model are rewritten in a way that they can be handled by four different usages of the tracking function.

The function takes between two and five parameters. The first one always determines which steps the function has to perform and the number and meaning of the following parameters. There are six possible values for it. Table 4.2 lists them and also explains their meaning. The according six calls are explained below.

| **Paramete value** | **Meaning** |
|---|---|
| ASSIGN_VP_PRE | A call before a compound assignment to a variable. The function backs up the data mapped to the left side. |
| ASSIGN_PROP_PRE | A call before an assignment to some property. The points-to relationship between owner and property value is removed. When indicated by the second parameter, the data mapping for the property value backed up. |
| ASSIGN_VP_VP | A call after an assignment of some variable to another variable. The mapping for the left variable has to be updated. |
| ASSIGN_VP_BOP | The result of a binary operation was assigned to some variable. The mapping for the variable must be updated based on the two operands of the binary operation. |
| ASSIGN_VP_PROP | The function updates the mapping for a variable after it got assigned the value of some property. |
| ASSIGN_PROP_VP | A call to update the mapping for a property which got assigned the value of some variable. The points-to mapping for the property owner is also updated to cover the new property value. |

Table 4.2: Possible values for the first parameter of the tracking function

**if4js_trackStatement("ASSIGN_VP_PRE", varName):**

This call only has one additional parameter. It is inserted prior to compound assignments to variables. The second parameter is the name of the left variable. The tracking function backs up the data mapping for its value, which in the model corresponds to $s_p(f(b, [lhs]))$

in the storage function update for *CompoundAssignment1*. After it has been overwritten, a subsequent tracking call will use it perform the storage function update for the left value.

**if4js_trackStatement("ASSIGN_PROP_PRE", append, objVar, nameVar):**

This call is inserted before any assignment to a property. The parameters *objVar* and *nameVar* hold the names of variables pointing to the owner and name of the property. They are used to dissolve the points-to mapping between owner object and property value, which is required for *Assignment3/4* and *CompoundAssignment2*. For compound assignments, *append* holds *true* and the call also backs up $s_p(f(b, [obj, prop]))$ for the left value like above.

**if4js_trackStatement("ASSIGN_VP_VP", append, leftName, rightName):**

This call updates the PIP after assignments. *LeftName* and *rightName* hold the names of the involved variables. The mapping for the left value is set to the data associated with the right value. If *append* is *true*, the backup data is added to the new mapping. Hence, the call belongs to the storage function update of either *Assignment1/2* or *CompoundAssignment1*, depending on the assignment operator and the *rhs* value type.

**if4js_trackStatement("ASSIGN_VP_BOP", append, leftName, lopName, ropName):**

Similar to the previous call, this one updates the PIP but after assignments having a binary operation as right value. In terms of the model, this is a combination of *BinaryOperation* with *Assignment1/2* or *CompoundAssignment1*. To track the occurred flows, the tracker uses the names of both involved operands in *lopName* and *ropName*.

**if4js_trackStatement("ASSIGN_VP_PROP", append, leftName, objVar, propVar):**

This call is used to decompose property expressions. The left variable of the respective assignment holds the property value afterwards. To track the associated data the PIP is updated for the variable named by *leftName*. The owner object is given by *objVar* and the property name by *propVar*. Again, *append* signals whether backup data has to be considered. So this is also a variant for the storage function update for *Assignment1/2* or *CompoundAssignment1*.

**if4js_trackStatement("ASSIGN_PROP_VP", append, objVar, propVar, rightName):**

Assignments to properties are handled by this type. As before *objVar* and *nameVar* identify the property. They are used to establish the relationship between the object and the new property value in the model, which is the missing part of the points-to function update in *Assignment3/4* and *CompoundAssignment2*. For the storage function update, *rightName* and *append* indicate which data has to be assigned in the PIP.

# 5 IF4JS - Implementation

**Clarification:** All statements given about the structure and architecture of V8 are mainly based on code reading and secondary sources such as mailing lists and blog posts. This is due to the fact that there are no detailed architecture documents about the internals of V8 available. Although the code contains detailed comments they do not reveal the full rationale behind the system design. Thus, explanations may be incomplete or not match the intentions of the developers.

Except for the XHR Interceptor, IF4JS is embedded into V8. Like Chromium, WebKit and V8, it is written in C++. The interceptor consists of a few lines of code that we added to the respective ObjectTemplate in WebKit. The remaining components are each realized by an own C++ class residing in V8. All classes are defined in the same header file *if4js.h*. To integrate the AST Rewriter into the compilation process of V8, we added some code to the compiler. It makes sure that ASTs are passed to the rewriter before they are compiled.

Furthermore, we extended V8's API, used by WebKit to communicate with it, by six methods. Three of them are required by the XHR Interceptor to communicate with PDP and PIP. One is used to register the tracking function in the JavaScript environment. Another one is required to tell IF4JS whether follow-up code has to be instrumented or not and the last one exists for debugging purpose.

Next, we describe how a development environment for IF4JS can be prepared and introduce where WebKit and V8 are located in Chromium. Afterwards, we describe the implementation and integration of IF4JS in detail.

## 5.1 Notes on Chromium, WebKit and V8

We integrated IF4JS into version r157275 of Chromium [24]. It is available as a patch for this revision. To set up a development environment, one should follow the instructions from the project website [17] and use the referenced archive instead of the latest version. Afterwards the patch bundled with this thesis has to be applied to integrate IF4JS into the code. Using an IDE eases navigation through the code a lot. A manual to setup an Eclipse project for Chromium is available [25].

V8 is located in the subfolder *v8/* of Chromium's main source directory. WebKit resides in the relative path *third-party/WebKit/*, the V8 Bindings can be found under *third-party/WebKit/Source/WebCore/bindings/v8/*. The implementation of IF4JS only affects V8 and the bindings, everything else remained unchanged.

### 5.1.1 Modifications in V8 Bindings

The V8 Bindings comprise all code that is required to connect WebKit and the DOM to V8. We made changes at three spots. Each modification carries the comment "IF4JS extension" to ease their localization.

On the one hand, the bindings contain management code. Particularly important in our context is the *ScriptController* class. It feeds DOM events received from WebCore into V8 for their evaluation. This is the place where we added code to inform IF4JS about incoming code that must be instrumented. Other management code, for instance, keeps track of open windows and tabs and the respective V8 contexts that belong to them.

On the other hand, there are a lot of Object- and FunctionTemplates (see Section 2.2.1). They provide the implementations for the DOM and XHR API. One of them is the template for the global object in V8 contexts. It reflects the window of the respective tab and is provided by the *V8DOMWindowShell* class. IF4JS extends its initialization code to register the tracking function *if4js_trackStatement*.

The third change affects the XHR implementation. There we added the code for the XHR Interceptor in the methods responsible to initialize and send requests.

As a side note, for the case one misses ObjectTemplates for the major part of the DOM API: Most templates are automatically generated during Chromium's build process from their interface definitions. Thus, they do not exist before Chromium is compiled for the first time. For a Debug build, the generated parts are created in */out/Debug/gen/webcore/bindings/* relative to Chromium's main directory.

### 5.1.2 Overview of V8

The organization of V8's source code does not reveal much about its architecture. Most header and source files are directly placed in the source directory *v8/src/*. Only the machine code generators have own subfolders according to their target platform. The code is distributed among two namespaces, *v8* and *v8::internal*. The embedder API resides in *v8* and serves as a mediator between the user and the core implementation in *v8::internal*. Knowing this is important, because some classes like *Object*, *String* or *Handle* exist in both, with different implementations, of course.

A good starting point to understand how V8 works is the API. By tracing the control flow of its methods, one can comprehend the performed steps. The API is defined in *v8/include/v8.h* and implemented in *v8/api.cc*. It provides the possibilities to create isolated runtime environments via *Context*, to parse and execute JavaScript code via *Script* and all types required to implement and register custom functionality. *Script::Compile* and *Script::Run* are the methods used by WebCore to execute direct JavaScript. DOM events are executed by calling custom functions. An elaborate guide through the control flow of *Script::Compile* can be found in [23].

## 5.2 PDP Interface

IF4JS has only one hard coded policy: No sensitive data may be sent away. To decide about it, the PDP class *IF4JS_PDPWrapper* provides one method that takes a request and its contained data. If there is any sensitive data, the request is prohibited. The complete code for the PDP is shown in Listing 5.1. Although currently not used, the request is passed to the PDP to leave room for more detailed checks based on any properties.

```
class IF4JS_PDPWrapper{
public:
  // Takes a request and the set of data associated with it.
  // Returns true if the request may be sent, false if not.
  static bool IsXhrSendAllowed(Handle<Value> request,set<string> data)
  {
      return data.size() == 0;
  };
};
```

Listing 5.1: The complete PDP implementation.

## 5.3 PIP Interface

The PIP component is realized by class *IF4JS_PIPWrapper*. Figure 5.1 shows its structure. It has only static members, simply for the sake of less code. So, future work could change this without hidden ramifications.

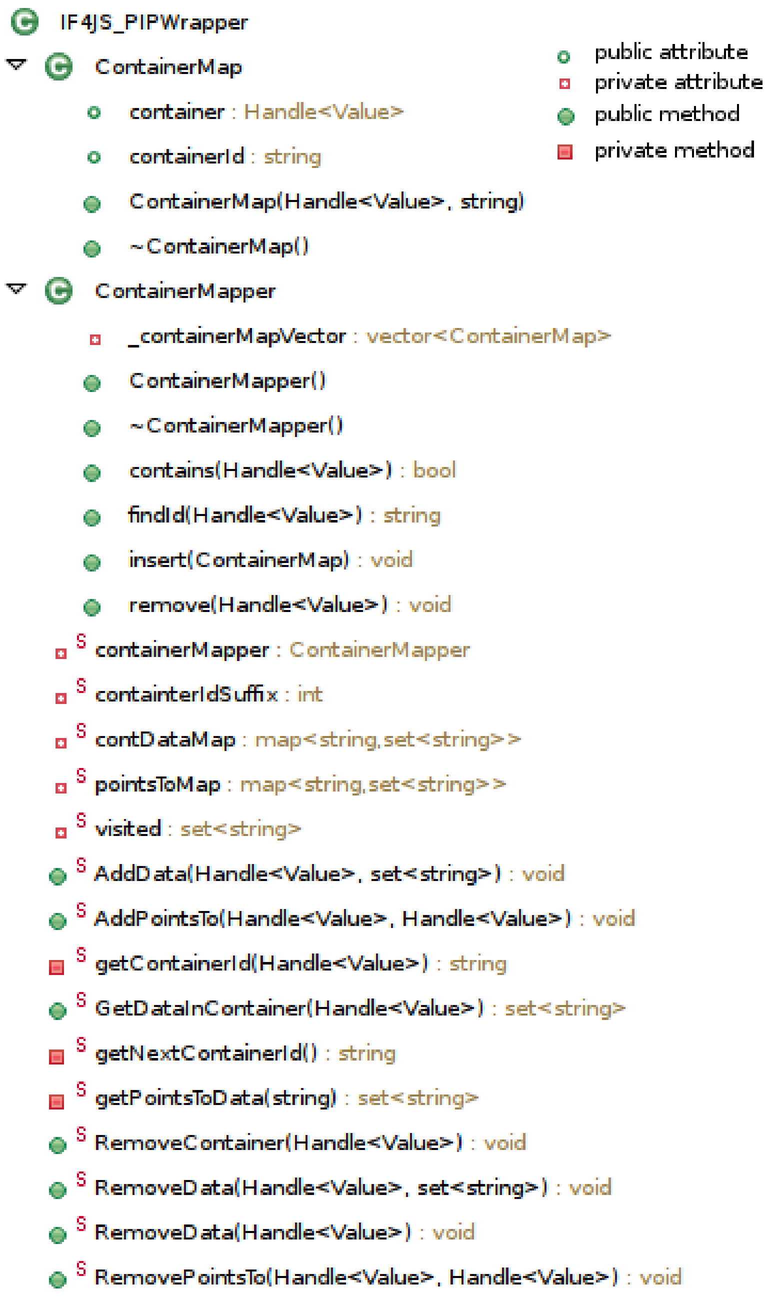
The wrapper has two nested classes, *ContainerMapper* and *ContainerMap*. They are used to associate containers with a string identifier. In JavaScript, values are containers and IF4JS identifies them by their references. However, references have no meaning outside of the runtime environment. As we developed IF4JS with multi-layer usage control in mind, we decided to already integrate a mapping for them.

The public methods of *IF4JS_PIPWrapper* provide the PIP interface introduced in Section 4.2.3. There is a second *RemoveData* method which only receives a container and no data set. When it is called, the mapping for the given container is cleared. This eases the synchronization between the data to container mappings in the DOM and the model instance maintained by IF4JS.

### 5.3.1 Mappings and their implementation

Besides the mapping between JavaScript values and container ids in *containerMapper*, the PIP implementation maintains two additional mappings. In *contDataMap*, container ids are mapped to the set of sensitive data contained in the respective container. This represents the storage function from the proposed model. Analogously, *pointsToMap* realizes the points-to function. Both mappings are implemented using the template class *map* from the standard library.

Unfortunately, it is not possible to also use *map* for the container id mapping, because it organizes keys relying on a strict weak order. The memory addresses of JavaScript values could not be used as ordering criterion as the garbage collector moves objects around and thus likely changes the order. *Handles* (compare 2.2.1), which wrap values, can not be used as keys either, because different handles may refer to the same value.



```
IF4JS_PIPWrapper
  ▽ ContainerMap
      ○ container : Handle<Value>
      ○ containerId : string
      ● ContainerMap(Handle<Value>, string)
      ● ~ContainerMap()
  ▽ ContainerMapper
      ▫ _containerMapVector : vector<ContainerMap>
      ● ContainerMapper()
      ● ~ContainerMapper()
      ● contains(Handle<Value>) : bool
      ● findId(Handle<Value>) : string
      ● insert(ContainerMap) : void
      ● remove(Handle<Value>) : void
  ▫ S containerMapper : ContainerMapper
  ▫ S containterIdSuffix : int
  ▫ S contDataMap : map<string,set<string>>
  ▫ S pointsToMap : map<string,set<string>>
  ▫ S visited : set<string>
  ● S AddData(Handle<Value>, set<string>) : void
  ● S AddPointsTo(Handle<Value>, Handle<Value>) : void
  ■ S getContainerId(Handle<Value>) : string
  ● S GetDataInContainer(Handle<Value>) : set<string>
  ■ S getNextContainerId() : string
  ■ S getPointsToData(string) : set<string>
  ● S RemoveContainer(Handle<Value>) : void
  ● S RemoveData(Handle<Value>, set<string>) : void
  ● S RemoveData(Handle<Value>) : void
  ● S RemovePointsTo(Handle<Value>, Handle<Value>) : void
```

| | |
|---|---|
| ○ | public attribute |
| ▫ | private attribute |
| ● | public method |
| ■ | private method |

Figure 5.1: The structure of *IF4JS_PIPWrapper* with its nested helper classes *ContainerMap* and *ContainerMapper*. The red S marks static members.

## 5.4 AST Rewriter

The rewriter is implemented in class *IF4JS_ASTRewriter*. Its methods *SetActive* and *IsActive* are used to control when the rewriter has to inject the runtime tracker. They serve as getter and setter for its private *_active* attribute.

Then there are 47 more public methods. They are all inherited from V8's *AstVisitor* class. This class provides the interface to implement a visitor for ASTs according to the visitor pattern. Among the prescribed methods there is one *VisitXYZ* method for each of the 35 expression and statement types in V8. However, the only method used to instrument functions is *VisitFunctionLiteral*. The rewriter has an empty implementation for the other visitor methods. Figure 5.2 shows the members of *IF4JS_ASTRewriter* but leaves out the methods from *AstVisitor* except for *VisitFunctionLiteral*.

### 5.4.1 Context and utilities

While processing a function, V8 maintains a *CompilationInfo* object to gather information about the function. For example, it contains the source code, the genrated AST, the context of the function and will also hold the generated machine code in the end.

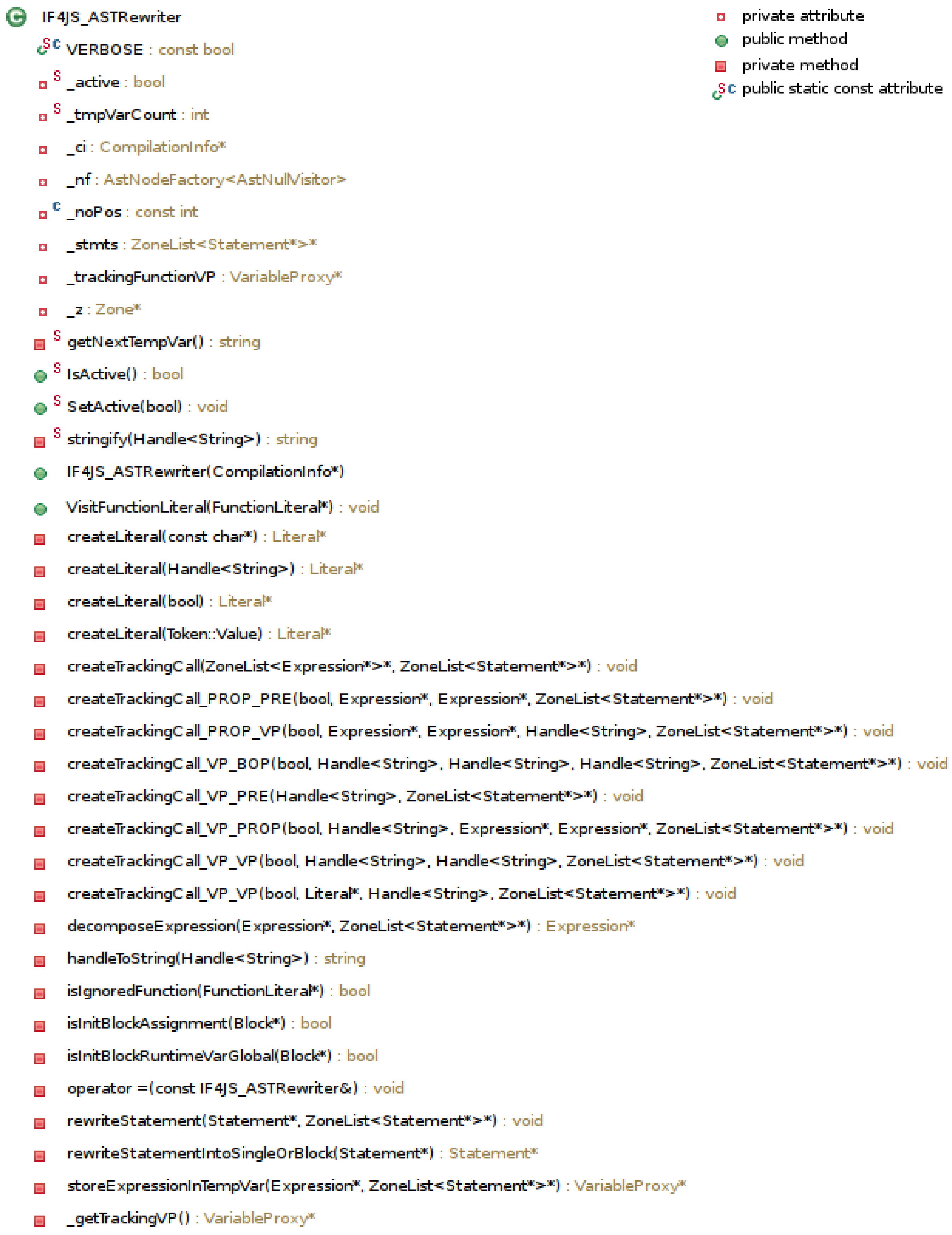
Each time a function has to be rewritten, the processing is interrupted right before V8 compiles the machine code. To perform the rewriting, a new *IF4JS_ASTRewriter* instance is created and given the described *CompilationInfo* object. From there it retrieves the AST that has to be instrumented. The statements created to replace the original function body are collected in *_stmts* which holds a V8 internal list data structure. To produce new statements and expressions, the rewriter uses an *AstNodeFactory* which also is obtained by the help of the *CompilationInfo* instance.

As explained in the design, a basic approach to track flows is the observation of assignments. Thus, the rewriter uses *storeExpressionInTempVar* to create statements that store expression values in new variables. Subsequent tracking calls then record the induced flow. This calls are created by the *createTrackingCall* methods. They produce *ExpressionStatements* which call the runtime tracker as introduced in Section 4.4.2. The suffixes of the method names indicate which calls they create. Both, the assignment creator and the tracking call generators receive a list as last parameter to which they add the created statements.

The *createLiteral* methods help to create the parameters for the tracking function. They derive expressions which hold the names of variables or the value of the *append* parameter.

### 5.4.2 A rewriting life-cycle

The rewriting process always starts with a call to *VisitFunctionLiteral* which receives the program or function to instrument as a pointer to a *FunctionLiteral* expression. This method, Listing 5.2 shows its implementation, simply passes each statement in the body to *rewriteStatement* (line 13). Afterwards it replaces the original body with the instrumented statements. As the method operates on a pointer to the original function, there is no need to change anything else. The *CompilationInfo* mentioned above also points to the same object and hence, the compilation process will compile the instrumented function without being aware of the modification.



```
IF4JS_ASTRewriter
    SC VERBOSE : const bool
    S _active : bool
    S _tmpVarCount : int
      _ci : CompilationInfo*
      _nf : AstNodeFactory<AstNullVisitor>
    C _noPos : const int
      _stmts : ZoneList<Statement*>*
      _trackingFunctionVP : VariableProxy*
      _z : Zone*
    S getNextTempVar() : string
    S IsActive() : bool
    S SetActive(bool) : void
    S stringify(Handle<String>) : string
      IF4JS_ASTRewriter(CompilationInfo*)
      VisitFunctionLiteral(FunctionLiteral*) : void
      createLiteral(const char*) : Literal*
      createLiteral(Handle<String>) : Literal*
      createLiteral(bool) : Literal*
      createLiteral(Token::Value) : Literal*
      createTrackingCall(ZoneList<Expression*>*, ZoneList<Statement*>*) : void
      createTrackingCall_PROP_PRE(bool, Expression*, Expression*, ZoneList<Statement*>*) : void
      createTrackingCall_PROP_VP(bool, Expression*, Expression*, Handle<String>, ZoneList<Statement*>*) : void
      createTrackingCall_VP_BOP(bool, Handle<String>, Handle<String>, Handle<String>, ZoneList<Statement*>*) : void
      createTrackingCall_VP_PRE(Handle<String>, ZoneList<Statement*>*) : void
      createTrackingCall_VP_PROP(bool, Handle<String>, Expression*, Expression*, ZoneList<Statement*>*) : void
      createTrackingCall_VP_VP(bool, Handle<String>, Handle<String>, ZoneList<Statement*>*) : void
      createTrackingCall_VP_VP(bool, Literal*, Handle<String>, ZoneList<Statement*>*) : void
      decomposeExpression(Expression*, ZoneList<Statement*>*) : Expression*
      handleToString(Handle<String>) : string
      isIgnoredFunction(FunctionLiteral*) : bool
      isInitBlockAssignment(Block*) : bool
      isInitBlockRuntimeVarGlobal(Block*) : bool
      operator =(const IF4JS_ASTRewriter&) : void
      rewriteStatement(Statement*, ZoneList<Statement*>*) : void
      rewriteStatementIntoSingleOrBlock(Statement*) : Statement*
      storeExpressionInTempVar(Expression*, ZoneList<Statement*>*) : VariableProxy*
      _getTrackingVP() : VariableProxy*
```

Figure 5.2: The structure of *IF4JS_ASTRewriter*. The S indicates static members.

```
void IF4JS_ASTRewriter::VisitFunctionLiteral(FunctionLiteral* funLit){
  // runtime tracker initialization
  v8::IF4JS_RuntimeTracker::InitializeMap();
  // Only rewrite functions if requested
  if (IF4JS_ASTRewriter::IsActive() && !isIgnoredFunction(funLit)){
    // fetch the function body
    ZoneList<Statement*>* body = funLit->body();
    // create the list to collect the instrumented statements
    _stmts = new (_z) ZoneList<Statement*>(body->length(), _z);
    // iterate over the function body and rewrite each statement
    for (int i = 0; i < body->length(); i++) {
      // the rewritten statements are put into the collector
      rewriteStatement(body->at(i), _stmts);
    }
    // replace the original function body with the instrumented one
    body->Clear();
    body->AddAll(_stmts->ToVector(), _z);
  }
}
```

Listing 5.2: The code of *VisitFunctionLiteral*.

The *rewriteStatement* method reflects the *instrument* function we used to specify the rewrite rules in Section 4.3.3. There we explained, how instrumented statements are derived from an input statement but not exactly where they are stored. Note that the rewrite method has a second parameter. It is a statement list named *collector* to which the rewritten statements are added. This list allows *rewriteStatement* to recursively rewrite statements without mixing up the nesting of statements. See Listing 5.3 for an illustration. When a block has to be rewritten, the method creates a new list (line 2) and lets the nested statements be rewritten into it without mixing up the list for the block (lines 4-7). Afterwards, the block is added to its intended list (line 11).

```
 1 // a block, we rewrite the sequence of nested statements
   ZoneList<Statement*>* rewBlockStmts =
 3                                   new(_z) ZoneList<Statement*>(3, _z);
   for (int i = 0; i < block->statements()->length(); i++){
 5   Statement* cStmt = block->statements()->at(i);
     this->rewriteStatement(cStmt, rewBlockStmts);
 7 }
   // replace original block statements with the rewritten ones
 9 block->statements()->Clear();
   block->statements()->AddAll(rewBlockStmts->ToVector(), _z);
11 collector->Add(block, _z);
```

Listing 5.3: Code from IF4JS which illustrates nested statement rewriting.

Also the *decompose* function from the rewrite rules has a counterpart in *IF4JS_ASTRewriter*: *decomposeExpression*. It splits up given expressions into several individually trackable statements and generates the required runtime tracking calls. *DecomposeExpression* also receives a list to store created statements. It returns the expression which serves as replacement for the handled expression.

## 5.5 Runtime Tracker

For the Runtime Tracker we first describe the implementation of the XHR Interceptor and then explain how the runtime tracking function evolves the model instance that is maintained in parallel to the tracked JavaScript code.

### 5.5.1 The XMLHttpRequest Interceptor

The XHR interceptor consists of two modifications in the *V8XMLHttpRequest* class. It is the C++ implementation of the object and function templates for *XMLHttpRequest*. *V8XMLHttpRequest::openCallback* and *::sendCallback* provide the implementations of the API functions *open* and *send* [26]. Their implementation can be found in */custom/V8XMLHttpRequestCustom.cpp* relative to the bindings directory. We added some code to both methods to implement the interceptor. As the interceptor needs to communicate with the PIP to perform its task, we added the methods *V8::PIPGetDataInContainer* and *V8::PIPAddDataToContainer* to the V8 API. They are required because the PIP is located in V8 and thus not directly accessible from WebKit.

**XMLHttpRequest.open:** The *open* method configures a XHR request. The parameters determine whether the request should use HTTP GET or POST and the URL it will be sent to. Because URLs may contain GET-parameters to transport data, we added the code in Listing 5.4 to check whether the URL parameter contains sensitive data. In the code, *args* is an array holding the parameters passed to *open*. If the URL contains sensitive data, the PIP is updated and the request marked to also contain the same data as the received URL.

```
// We check for sensitive data in the URL
set<string> urlData = v8::V8::PIPGetDataInContainer(args[1]);
if (urlData.size() > 0){
  // Update the mapping for the opened request
  set<string> reqData;
  reqData.insert(urlData.begin(), urlData.end());
  if(v8::V8::IsIF4JSVerbose()){
    cout << "__ XHR created with sensitive data" << endl;
  }
  v8::V8::PIPAddDataToContainer(args.Holder(), reqData);
}
```

Listing 5.4: The code used to track sensitive data when XHRs are opened

**XMLHttpRequest.send:** To start a request, the *send* method is used. It takes an optional parameter which for POST requests provides the POST data to send along. How a request is performed and where it is directed to depends on its configuration. Before *send* can be called, the request must have been configured via *open*. Hence, when a request should be sent, we must check whether it already contains sensitive data and if *send* received sensitive POST data. Listing 5.5 shows the required code. If there is sensitive data somewhere, the interceptor terminates the sending call (line 14) before any data left the system.

```
v8::Local<v8::Object> request = args.Holder();
// Fetch data already present in the request
set<string> reqData = v8::V8::PIPGetDataInContainer(request);
// Next add sensitive data from the first parameter
if (args.Length() >= 1){
  set<string> postData = v8::V8::PIPGetDataInContainer(args[0]);
  reqData.insert(postData.begin(), postData.end());
}
// Now we query the PDP whether we may send the request or not
if(!v8::V8::IsXhrSendAllowed(request, reqData)){
  if (v8::V8::IsIF4JSVerbose()){
    cout << "__ XHR with sensitive data ... aborted" << endl;
  }
  return v8::Undefined();
}
```

Listing 5.5: The code used to detect whether XHRs are about to send sensitive data.

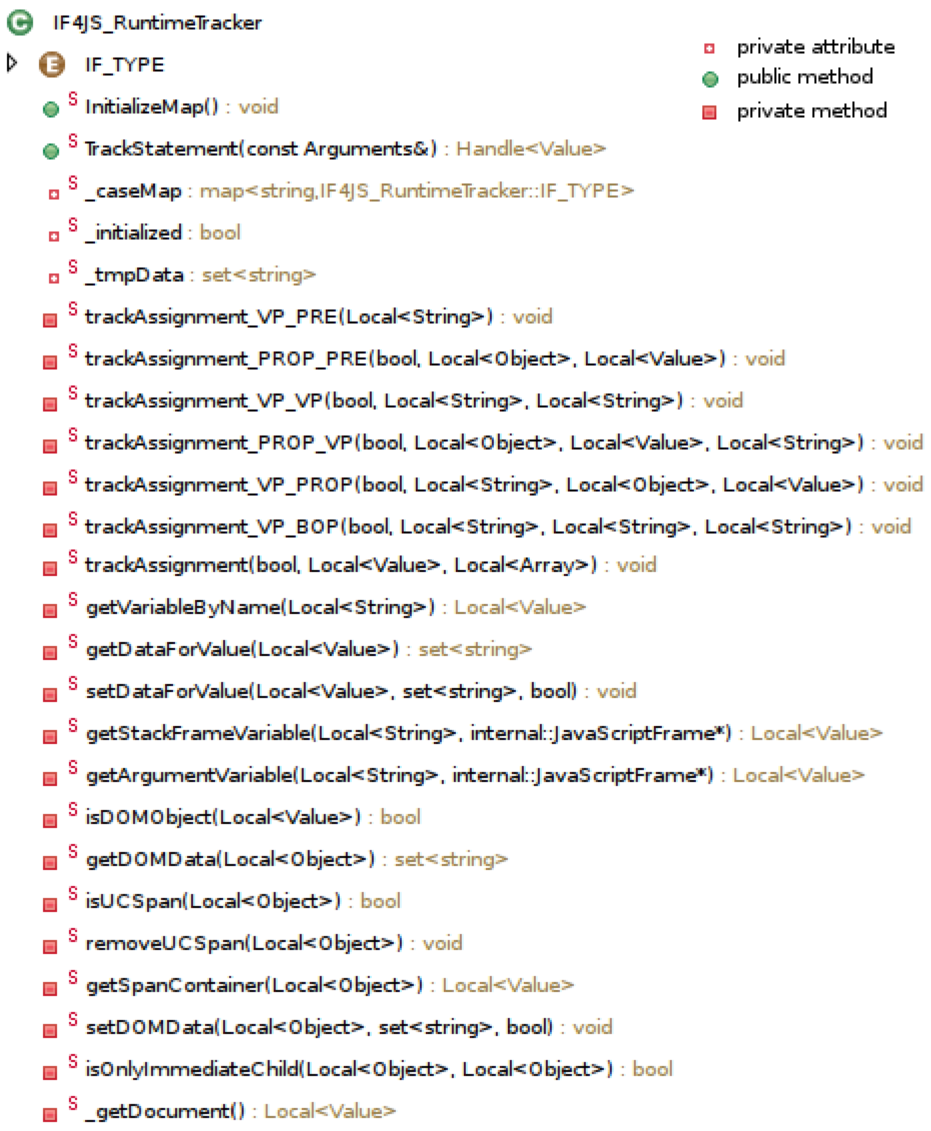
### 5.5.2 The runtime tracking function

Actually, the runtime tracking function is not only just one method or function. It subsumes the functionality provided by *IF4JS_RuntimeTracker*. We referred to it as the tracking function, because all its activity is triggered via injected calls to *if4js_trackStatement*. IF4JS register the method *TrackStatement* of the runtime tracker class as callback to handle *if4js_trackStatement* calls. Figure 5.3 reveals *IF4JS_RuntimeTracker*'s structure.

Besides evolving the storage and points-to functions maintained in the PIP, it also takes care of connecting IF4JS with the DOM. When tracked variables refer to DOM nodes, usage control information is synchronized between the DOM and IF4JS. If an expression reads data from a DOM node, the tracker inspects whether there is an outer usage control <span> tag in the ancestor chain of the node and, if so, associates the data specified in the <span> with the node value in the runtime environment. Analogously, if an expression pushes data into a DOM node, *IF4JS_RuntimeTracker* makes sure to update the DOM with the associated usage control information. This either means it adds data to a usage control <span> or creates a new one if there was none before. In Figure 5.3, the eight involved methods can be seen in the lower part (from *isDOMObject* to *_getDocument*). Elaborate comments in the respective code give insights about the concrete steps performed.

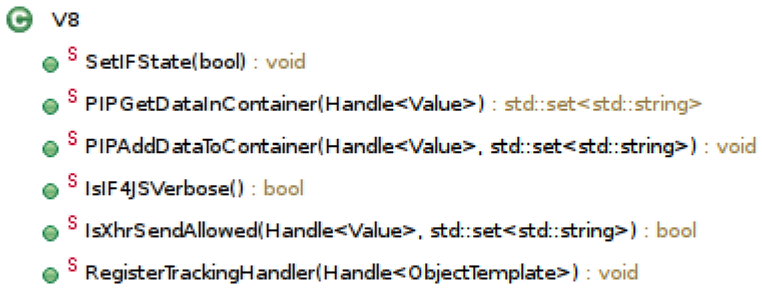
As said before, *TrackStatement* is the callback for *if4js_trackSatement*. Its task is to delegate the incoming calls according to the given parameters. For each of the six calls described in Section 4.4, there is an according *trackAssignment_* call which performs the specified actions. The two calls ending on '_PRE', which back up previous left side data if required, use *_tmpData* to store the backup. Any subsequent tracking call will look up the data and then clear the temporary store.

When the tracker performs its updates to the storage and points-to function, it must take care whether the involved variables are local, arguments or global. As the rewriter does not add hints about variable types, the runtime tracker must determine them by itself. This task is performed by *getVariableByName* which uses *getStackFrameVariable* and *getArgument-Variable* as helpers to lookup local variables and formal argument names. Determining the



```
IF4JS_RuntimeTracker
  IF_TYPE
  S InitializeMap() : void
  S TrackStatement(const Arguments&) : Handle<Value>
  S _caseMap : map<string,IF4JS_RuntimeTracker::IF_TYPE>
  S _initialized : bool
  S _tmpData : set<string>
  S trackAssignment_VP_PRE(Local<String>) : void
  S trackAssignment_PROP_PRE(bool, Local<Object>, Local<Value>) : void
  S trackAssignment_VP_VP(bool, Local<String>, Local<String>) : void
  S trackAssignment_PROP_VP(bool, Local<Object>, Local<Value>, Local<String>) : void
  S trackAssignment_VP_PROP(bool, Local<String>, Local<Object>, Local<Value>) : void
  S trackAssignment_VP_BOP(bool, Local<String>, Local<String>, Local<String>) : void
  S trackAssignment(bool, Local<Value>, Local<Array>) : void
  S getVariableByName(Local<String>) : Local<Value>
  S getDataForValue(Local<Value>) : set<string>
  S setDataForValue(Local<Value>, set<string>, bool) : void
  S getStackFrameVariable(Local<String>, internal::JavaScriptFrame*) : Local<Value>
  S getArgumentVariable(Local<String>, internal::JavaScriptFrame*) : Local<Value>
  S isDOMObject(Local<Value>) : bool
  S getDOMData(Local<Object>) : set<string>
  S isUCSpan(Local<Object>) : bool
  S removeUCSpan(Local<Object>) : void
  S getSpanContainer(Local<Object>) : Local<Value>
  S setDOMData(Local<Object>, set<string>, bool) : void
  S isOnlyImmediateChild(Local<Object>, Local<Object>) : bool
  S _getDocument() : Local<Value>
```

Figure 5.3: The structure of *IF4JS_RuntimeTracker*. All members are static.



```
V8
  S SetIFState(bool) : void
  S PIPGetDataInContainer(Handle<Value>) : std::set<std::string>
  S PIPAddDataToContainer(Handle<Value>, std::set<std::string>) : void
  S IsIF4JSVerbose() : bool
  S IsXhrSendAllowed(Handle<Value>, std::set<std::string>) : bool
  S RegisterTrackingHandler(Handle<ObjectTemplate>) : void
```

Figure 5.4: The six methods which extend the V8 API.

variable type is based on the assumption that local variables always have precedence before arguments which in turn precede global variables. So, *getVariableByName* checks one after another and returns the according value as soon as the given name was successfully resolved. This way, variable shadowing is covered. If the name could not be resolved to any type, the variable does not exist and the method returns *undefined*.

## 5.6 Integration into WebKit and V8

We now explain how the different parts of IF4JS are embedded into Chromium. There are four parts we had to change to integrate it, two in WebKit and two in V8. First we will describe the extensions of the V8 API. Then we explain how the tracking function is registered in the runtime environment and where the rewriter is informed about DOM events. Lastly, we show how the compilation process in V8 is intercepted and redirected through the rewriter.

### 5.6.1 V8 API extensions

The six methods extending the API are depicted with their signatures in Figure 5.4.

**IsIF4JSVerbose:** This method does not contribute to the actual information flow tracking. It is used to determine whether the modifications in WebKit should output debugging information to the console.

**SetIfState:** The code signaling DOM events from WebKit uses this to inform the rewriter about upcoming events. It is a wrapper which makes *IF4JS_ASTRewriter::SetActive* accessible through the API.

**RegisterTrackingHandler:** This registers *TrackStatement* from *IF4JS_RuntimeTracker* in the received object template. It is called with the template for the global object when a website is loaded.

**PIPGetDataInContainer and PIPAddDataToContainer:** This two functions let external code query the PIP to receive and update the data associated with JavaScript values. It is used by the XHR Interceptor.

**IsXhrSendAllowed:** This method performs a query for XHRs to the PDP. Given a request and the set of contained data, it returns whether the request may be sent or not.

### 5.6.2 Tracking function registration

The *V8DOMWindowShell* class mentioned above, which provides the global object within a browser window, is implemented in *V8DOMWindowShell.cpp*. The method *createContext* is in charge of configuring its template before it is registered in V8. To add the tracking function, we use one line of code which passes the prepared object template to the new V8 API function *RegisterTrackingHandler*.

### 5.6.3 Signaling DOM events

WebKit uses the *ScriptController* class to execute JavaScript. It is implemented in *Script-Controller.cpp*. For DOM events, after some preparation, the controller uses *callFunction-WithInstrumentation* to pass the event to V8. We added one line of code at the beginning of this method to call *SetIFState* from the V8 API with *true*. This activates the rewriter to instrument the upcoming event code. Before the method returns, we call *SetIFState* again to suspend the rewriter.

### 5.6.4 Redirecting the compilation process

To instrument functions, we need to intercept V8's compilation process right before the created AST is actually compiled into machine code. The respective code is located in *v8/compiler.cc*. V8 only compiles functions when they are executed afterwards. If some code contains a function definition, this function is just preprocessed. It will be compiled only if later on some other code, a DOM event for instance, actually calls this function. This principle is referred to as lazy compilation.

To cover both cases, direct and lazy compilation, we had to modify two spots in *compiler.cc*. One is located in the function *MakeFunctionInfo*, the other in *Compiler::CompileLazy*. The code, which is also marked with the comment "IF4JS extension", checks whether the rewriter is currently active and if so, creates a new instance of it. The created rewriter is then given to the processed function as a visitor what leads to *VisitFunctionLiteral* being called on it. This triggers the rewriting process as described above.

# 6 Evaluation

In this chapter we will evaluate the proposed model and the developed prototype. We look at the performance overhead caused by IF4JS and also define completeness and correctness as quality criteria for information flow tracking models.

## 6.1 Performance analysis

We created a small benchmark to asses the performance overhead caused by IF4JS. Our benchmark is based on the SunSpider test suite [27]. It aims at comparison of JavaScript implementations. The focus is on core JavaScript language features, hence, it does not involve the DOM or other additional browser APIs. This makes it well-suited for our purpose as we primarily modified V8. SunSpider consists of 26 single tests that are grouped in nine categories. They cover a wide range of scenarios ranging from string processing over mathematics and 3D graphics to crypto algorithms.

Executing the complete SunSpider benchmark with IF4JS was not possible in a reasonable time frame. Thus, we decided to execute single tests of the suite and use their results to estimate the overhead. We randomly chose five tests, each from a different category. Although it is a small subset of the benchmark, it is sufficient to highlight the dimension of the caused overhead.

### 6.1.1 Test setup

The tests we used are: *3d-cube*, *access-binary-trees*, *controlflow-recursive*, *crypto-md5* and *string-base64*. Unless otherwise stated, the result for a particular test case is given by the averaged duration of five individual runs. Each run was conducted in a new instance of the browser. The DVD belonging to the thesis contains the test cases and raw results.

The benchmark was performed on a Dell Precision T5500 with an Intel Xeon E5645 CPU, 32 GB DDR3 RAM and a solid state disk as system drive. Despite the power of the used hardware, we would expect similar results on a today's consumer PC, as Chromium has no high hardware requirements and the performance of JavaScript execution mainly depends on the capabilities of a single CPU core.

While executing the tests, we found out that IF4JS is not compatible with V8's optimizer Crankshaft. It dynamically changes executed code to speed-up execution. We will come back to this in the conclusion. To see Crankshaft's influence on our benchmark, we measured the unmodified Chromium versions once with and without it.

In total, we measured each test in six configurations: IF4JS, Chromium with and without Crankshaft, each of them as debug and release builds.

| Test | Type | w/ Crankshaft | w/o Crankshaft | | IF4JS | |
|---|---|---|---|---|---|---|
| 3d-cube | release | 14.4 | (x1.39) | 20 | (x89,953) | *1,295,330 |
| | debug | 248,6 | (x0,99) | 248 | | - |
| access-binary-trees | release | 4 | (x1.15) | 4.6 | (x1,819) | 7,276 |
| | debug | 30 | (x1.00) | 30 | (x4,285) | 128,558 |
| controlflow-recursive | release | 3.4 | (x1.41) | 4.8 | (x927) | 3,153 |
| | debug | 16.2 | (x1.01) | 16.4 | (x6,138) | 99,443 |
| crypto-md5 | release | 6 | (x0.97) | 5.8 | (x269,800) | *1,618,799 |
| | debug | 67.4 | (x0.63) | 42.2 | | - |
| string-base64 | release | 8.6 | (x0.77) | 6.6 | (x8,359) | 71,885 |
| | debug | 79.8 | (x1.00) | 79.8 | (x11,412) | *910,656 |

Table 6.1: The results of our benchmark. All figures describe runtime in milliseconds. In each row, the coefficients in brackets are related to the result with Crankshaft enabled. For values marked with * we measured only one run. Configurations marked with '-' have not been completed, see 6.1.2 for explanations.

### 6.1.2 Test results

The benchmark results can be found in Table 6.1. We didn't complete two tests, because of the supposed execution time. Assuming the factors for the release builds to be tolerably applicable, a run with IF4JS in debug mode would take more than 6 hours for *3d-cube* and more than 5 hours for *crypto-md5*.

The results clearly render IF4JS unfeasible for scenarios with computationally intensive tasks. In summary, the overall performance overhead due to rewriting and runtime tracking lies in the range of three to five orders of magnitude. This is not only a matter of proportionality. The fastest test case for IF4JS takes about 3 seconds which creates a perceivable delay, not to speak of nearly 27 minutes for *crypto-md5*.

The major overhead is caused by runtime tracking. Rewriting is only performed once for each executed function, independently of the numbers of invocations. It has a complexity of $O(n)$ in terms of AST nodes, as each node is visited once. But the injected tracking calls are executed for each invocation. Among the tracked expressions, calls and nested expressions cause the most overhead. For calls, each parameter is stored in a separate variable which is tracked for flows. This at least adds two statements per parameter, one of which is a tracking call. Nested expressions are decomposed similarly. Each sub expression is tracked via an assignment to a temporary variable, also creating at least two additional statements.

The MD5 test has many function invocations, with up to seven parameters, and uses binary operations with up to ten sub expressions, which explains the immense slow down compared to the control flow test. The latter is still nearly 1000 times slower than in the optimal case, but the overhead lies within three instead of five orders of magnitude while the MD5 calculation in the unmodified Chromium only takes twice the time of the control flow test. Having a closer look reveals that the control flow test uses less nesting in expressions and significantly less calls, which also have fewer parameters.

### 6.1.3 Further experiments

We suspect the tracking calls to be the primary reason for the produced overhead. To verify this, we compiled IF4JS with commenting out the line that inserts the tracking calls into rewritten ASTs. A short test with *3d-cube* (26 ms/283 ms) and *crypto-md5* (9 ms/50 ms) confirms our assumption. The gained insight through this little experiment is not surprising, but it affirms that future work should focus on this issue.

A first question would be whether it is possible to improve the performance of tracking calls to a satisfying level by optimizing the used data structures and algorithms, as we never considered their performance during implementation, or if the context switches alone already create too much overhead. To get a first impression, we made another experiment. This time, the tracking calls were inserted again, but they immediately returned without actually performing any tracking. Based on the results for *crypto-md5* (56 ms/2960 ms) and *3d-cube* (129 ms/6190 ms), there is a high potential for improvement in the implementation. At least for release builds it could be possible to reach or approximate acceptable performance.

We suppose the PIP implementation could be dramatically improved by replacing copy assignments with expressions using references or pointers. Also looking up container ids could be accelerated. Currently the list of containers is traversed twice, once to check whether a container is already known and if so once again to fetch it. Using an out parameter for the lookup result could save the second traversal. It would be useful to profile IF4JS to find out the greatest bottlenecks in the implementation and concentrate on them first.

### 6.1.4 Field testing

Regardless of the bad benchmark results we were interested in the perceived performance of IF4JS on regular websites. Websites usually cause little load compared to the benchmark so they could possibly still be usable. To get an impression of IF4JS in the wild we surfed the top six Alexa websites [28] with a release build of IF4JS. At the time of writing these were: Google, Facebook, Youtube, Yahoo, Baidu and Wikipedia [29–34].

On Wikipedia we did not perceive any delays and articles were displayed correctly. This is likely due to mainly static content. However, the overlay with a call for donations currently displayed on top of each article did not show up. Youtube also did not appear to be slower, but for most videos the preview pictures were not displayed. Also Google's web search only partly worked. The regular search displayed results correctly but the autocomplete feature, which displays similar and popular search terms just-in time as a drop-down menu, was not shown. The image search did not display any results and always showed the message that no matching images could be found. In Facebook we could login but we did not get any content displayed which is usually dynamically created by JavaScript. On the pages of Yahoo and Baidu the active tab simply crashed because of exceptions in V8.

This suggests wrong or incomplete rewriting rules. We guess that they are rather incomplete than wrong as we created examples which worked as expected. These examples of course only use statement and expression compositions we anticipated in the implementation. In the web JavaScript often is compressed using tools like Google's Closure Compiler [35]. It analyzes JavaScript code and rewrites it to be as small as possible. A construct often used for this is the comma-operator. In V8 this is a binary operator which behaves different from the action we modeled as *BinaryOperation*. Hence, it is most likely treated wrong by the current implementation. As optimizers often use it to initialize variables it could be an explanation for the erroneous behavior on Wikipedia, Youtube, Google and Facebook. We need to investigate the JavaScript code in these websites in detail and compare it with the results of IF4JS' rewriting process in order to fix the observed issues.

## 6.2 Completeness and correctness

Whether an information flow tracking approach detects all occurring flows, and also if each indicated flow has an actual counterpart in the tracked system, are important criteria to asses its practicability in a usage control system. We refer to these two criteria as *completeness* and *correctness*. We define and discuss them speaking of models and their properties, but the actual implementations also play a role. Thinking of our solution, the model alone would clearly not be able to capture many flows in JavaScript. But the way IF4JS rewrites tracked code allows to capture also flows for constructs that are not directly reflected in the model.

**Completeness criterion:** Completeness demands that actually occurring flows in the monitored system must be captured by a model. However, it does not consider false positives. Thus, a model indicating flows which did not occur can still be *complete* as long as it also captures all actual flows. But high over-approximation significantly contributes to the label-creep problem. This hinders fine-grained usage control, which essentially is the goal of using cross-layer approaches.

**Correctness criterion:** To evaluate a model with respect to over-approximations, we use a notion of *correctness*. We consider a captured flow to be *correct*, if there is a corresponding action in the observed system which actually caused this flow. Consequently, a *correct* model, which only tracks *correct* flows, would not produce false-positives. But, to be correct, a model does not necessarily have to track all occurring flows.

An ideal model would track at the same time *at least* but also *at most* all occurring flows. Thus, both criteria together describe the functional requirements for an ideal data flow tracking model. Our solution is not yet complete but the goal of future work should be to achieve completeness. We think the model could be correct for the parts covered by the ECMAScript specification though, because it tracks information flow on the level of single primitive values and thus for example also can track array fields. But we are not able to give formal evidence here. To provide a proof we would require a formal model of the ECMAScript language or its implementation in V8 which we could relate to our model. Unless we have such a model we can only present informal proof ideas for both criteria.

### 6.2.1 Proof ideas

**Completeness:** To show completeness, one must show that a solution *at least* covers all flows that occur in the tracked code. This could be done be a complete induction over the sequence of actions induced by the execution of JavaScript programs. By actions we mean changes in the JavaScript environment caused by the evaluation of expressions or execution of statements.

A JavaScript program is given by a sequence of statements. As explained in 4.3.1, statements themselves can have sub statements and consist of expressions. Adding one statement to a program could in fact add an arbitrary number of sub statements as for *Blocks*. This makes it difficult to perform an induction over the length of a program. But, as we are only interested in explicit flows, we can look at the sequence of actions caused by the execution of a program. The underlying control flow is transparent to this sequence, but for explicit flows it does not matter under which condition an action happens.

An induction over the sequence of actions would use an empty sequence as base case which changes nothing in a state and hence is trivially covered. The induction hypothesis would state that if a sequence of actions of length $n$ is covered, then the model also covers sequences of length $n + 1$. In the induction step we would have to show for any added action that the model covers the induced flow.

Due to the rewriting process applied by IF4JS, which would have to be proven as semantic preserving separately, the set of actions that could be added is limited to a certain subset of expression and statements in V8. This would be either actions that do not induce any flow, like *break* or *continue*, or assignments, calls or return expressions that can be directly mapped to action in the proposed model. For each case an argument had to be provided which proves why the flow is covered. As an example, for assignments from a variable to another variable we could argue that we can lookup the containers and associated data by the names of the involved variables and thus track the occurring flow.

**Correctness:** To show correctness, one must prove that any flow indicated by the model has a corresponding actual flow in the tracked code. Here a proof could show that information flow tracking is correct by construction. IF4JS detects flows by observing assignments in tracked code. So, if IF4JS detects a flow, this means that there also is an corresponding actual flow in the code. As the tracking is performed on the level of single properties and primitive values, there is no over-approximation here, at least for built-in types. If every single tracked flow from one container to another was correct, this would imply that also any flow described by more than one intermediary step was correct as it could only be composed from correct steps.

Unfortunately, this only holds for built-in objects and values. How objects of types added via object templates behave internally would have to be examined separately. This means, for a correctness proof also the implementation of the DOM API has to be examined which was not a subject of this thesis.

# 7 Conclusion and future work

In this thesis we presented an approach for information flow tracking in JavaScript and showed a prototype implementation of the proposed model in the Chromium browser. We provide two working examples in the *examples* folder of the submitted DVD. The introduced approach turned out to be feasible in general, but the evaluation revealed that our implementation as is produces intolerable overhead. Furthermore, we found out that currently not all statements and expressions are rewritten in a way that preserves the original semantics. Nevertheless, we believe that future work could fix these issues and improve it to cover the complete set of statements and expressions. We also see chances to optimize the implementation to an acceptable performance level. In the following we give an outlook on open problems that future work has to deal with.

## 7.1 AST rewriting

In terms of AST rewriting there are two major problems to be investigated for one of which it is unclear whether it can be solved with reasonable effort. The first problem concerns the incomplete rewriting that leads to the observations described in 6.1.4. The problem is caused by an incomplete handling of statements and expressions in the implementation and not about extending the implementation for completely unhandled ones. Ideally, the rewriting process injects the tracking code for covered constructs and changes nothing about unknown constructs. Obviously this is not the case. But it can be solved by identifying the constructs that we missed in the design or implementation and as a consequence are rewritten wrongly.

The second issue is to extend IF4JS to cover all statements and expressions in V8. As indicated in Section 4.3, new nodes cannot be easily generated for all statements and expressions. To fully rewrite the uncovered control flow statements, like for instance loops, it is necessary to understand how V8 captures jump labels and targets. This could be found in the AST or its nodes directly, or possibly also somewhere in the *CompilationInfo* data structure mentioned in Section 5.4. With the understanding about the control flow information it should be possible to arbitrarily rewrite also these AST nodes. Perhaps, another way to solve this issue could be to directly modify nodes by their pointers instead of replacing them by newly created nodes. As the original nodes would still exist, the control flow information collected by V8 would not be corrupted and thus could still work as expected. To do the proposed modifications of AST nodes it would be required to extend their classes by methods that provide write access to their nested nodes.

## 7.2 Performance improvements

In 6.1.3 we already gave hints on where and how the performance of the current implementation could be enhanced. But besides the pure implementation also the overall approach could be extended.

In Section 1.4 about related work we mentioned an information flow analysis approach combining static and dynamic analysis [4]. There program dependence graphs are used to statically analyze data and control dependency between statements in the handled code. The work referred to, done by Hammer et al. for WebKit's original JavaScript engine JavaScript Core, creates such graphs for each function to deal with implicit flows. Covering implicit flows is not possible in Chromium and V8, due to the lazy compilation approach and the fact that JavaScript execution, if possible, already starts before all code was loaded.

But it should be possible to statically calculate information flow graphs for functions that do not use dynamic code generation. Using them to track information flows in the respective functions could significantly reduce the amount of required runtime tracking calls. We guess that this approach could seriously speed up computations like the MD5 test from SunSpider which only depends on the given input. If the overall flows in a function are known there is no need to anymore track local variables that will not leave the scope of that function.

# Bibliography

[1] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrve. Document object model (dom) level 3 core specification. W3C Recommendation, April 2004.

[2] P. Wenz. Data Usage Control for ChromiumOS. Diploma thesis, Certifiable Trustworthy IT Systems (IPD), KIT Karslruhe, 2012.

[3] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *CSF*, pages 3–18, 2012.

[4] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM.

[5] Jonas Magazinius, Ro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. In *In Proc. IFIP International Information Security Conference*, 2010.

[6] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[7] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. Representation-independent data usage control. In *Proceedings of the 6th international conference, and 4th international conference on Data Privacy Management and Autonomous Spontaneus Security*, DPM'11, pages 122–140, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] R.S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40 –48, September 1994.

[9] Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens-Michael Kuhn. Distributed data usage control for web applications: a social network implementation. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 85 –96, New York, NY, USA, 2011. ACM.

[10] M. Harvan and A. Pretschner. State-Based usage control enforcement with data flow tracking using system call interposition. In *Network and System Security, 2009. NSS '09. Third International Conference on*, pages 373 –380, October 2009.

[11] Alexander Pretschner, Matthias Büchler, Matus Harvan, Christian Schaefer, and Thomas Walter. Usage control enforcement with data flow tracking for x11. In *5th International Workshop on Security and Trust Management (STM 2009)*, 2009.

[12] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[13] v8 - V8 JavaScript Engine. http://code.google.com/p/v8/.

[14] Chromium - The Chromium Projects. https://sites.google.com/a/chromium.org/dev/Home.

[15] Chromium blog - Google Chrome, Chromium, and Google. http://blog.chromium.org/2008/10/google-chrome-chromium-and-google.html.

[16] The WebKit Open Source Project. http://www.webkit.org/.

[17] For Developers - The Chromium Projects. https://sites.google.com/a/chromium.org/dev/developers.

[18] Design Elements - Chrome V8. https://developers.google.com/v8/design.

[19] Wingolog - posts tagged "v8". http://wingolog.org/tags/v8.

[20] Embedder's Guide - Chrome V8. https://developers.google.com/v8/embed.

[21] How WebKit Works - presentation by Adam Barth on October 30, 2012. https://docs.google.com/presentation/pub?id=1ZRIQbUKw9Tf077odCh66OrrwRIVNLvI_nhLm2Gi__F0#slide=id.p.

[22] Document Object Model (DOM) Level 3 Events Specification. http://www.w3.org/TR/DOM-Level-3-Events/.

[23] Jonas' weblog - "A script's tale - exploring V8". http://blog.peschla.net/2012/10/exploring-v8/.

[24] Chromium source code, revision 157275. http://chromium-browser-source.commondatastorage.googleapis.com/chromium.r157275.tgz.

[25] Chromium source code, revision 157275. http://code.google.com/p/chromium/wiki/LinuxEclipseDev.

[26] XMLHttpRequest - Living Standard as of 22th November 2012. http://xhr.spec.whatwg.org/#toc.

[27] SunSpider JavaScript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[28] Alexa Top 500 Global Sites. http://www.alexa.com/topsites.

[29] Google. https://www.google.de/.

[30] Facebook. https://www.facebook.com/.

[31] Youtube. https://www.youtube.com/.

[32] Yahoo. http://www.yahoo.com/.

[33] Baidu. http://www.baidu.com/.

[34] Wikipedia. http://en.wikipedia.org/.

[35] Closure compiler - a javascript optimizing compiler. http://code.google.com/p/closure-compiler/.