Analyse statique de code dynamique

Pierre Wilke

Stage de Master Recherche

Rapport de stage

Analyse statique de code dynamique

Auteur :
Pierre Wilke

Encadrants :
Frédéric Besson
Thomas Genet
CELTIQUE

Table des matières

Résumé

JavaScript est un langage de scripts utilisé sur la majorité des sites web modernes. Plusieurs plateformes permettent à présent de faire interagir différents *gadgets* JavaScript afin de fournir un contenu plus riche. Le langage JavaScript donne beaucoup de flexibilité aux développeurs et est extrêmement dynamique. Ces atouts viennent au prix d'une sécurité difficile à garantir. JavaScript propose notamment une fonction `eval` qui permet d'évaluer dynamiquement du code JavaScript à partir d'une chaîne de caractères. Cette fonction en particulier rend l'analyse statique difficile. La littérature propose quelques tentatives d'analyses statique ou dynamique pour traiter, partiellement, cette fonction qui reste un challenge. Cette fonction est un défi car elle demande l'analyse d'un ensemble de programmes possibles, alors que les analyses standards ne s'intéressent qu'à un unique programme. Afin d'attaquer ce défi, nous proposons une sémantique de JavaScript exprimée à l'aide de systèmes de réécriture de termes qui permet de raisonner sur des ensembles de programmes et qui se prête bien à l'analyse statique des états accessibles par ces programmes. Nous montrons comment cette sémantique peut être utilisée en prouvant automatiquement qu'une propriété est vérifiée pour un ensemble donné de programmes.

Mots-clés : analyse statique, JavaScript, méthodes formelles, code dynamique, systèmes de réécriture

Introduction

JavaScript est utilisé sur une grande majorité de sites web actuels, afin de dynamiser leurs pages. Il permet en effet d'effectuer des calculs, d'actualiser des informations, et d'interagir avec le site sans avoir à recharger la page, ce qui donne une plus grande fluidité à la navigation.

Le langage JavaScript est extrêmement dynamique, puisqu'en plus de pouvoir changer la relation d'héritage au cours de l'exécution, tout est mutable en JavaScript, notamment les objets natifs. On peut par exemple redéfinir le comportement de l'objet de base `Object`, rien qu'en modifiant son prototype. On peut également agir sur l'objet global (nommé `window` quand JavaScript est exécuté dans un navigateur, non accessible directement dans un interpréteur) et sur n'importe quel objet pour lui ajouter, modifier ou supprimer des propriétés lors de l'exécution.

À travers l'objet global, JavaScript peut accéder à la page HTML laquelle le script est inséré, à travers le DOM (*Document Object Model*). On peut accéder aux différents éléments HTML et ainsi écrire ou lire la page web, ce qui donne une grande puissance à JavaScript.

JavaScript offre une grande flexibilité aux développeurs, mais cela vient au prix d'un certain nombre de vulnérabilités. La plus commune est connue sous le nom de *cross-site scripting (XSS)*. Il s'agit de profiter d'un programme JavaScript vulnérable pour lui faire exécuter du code arbitrairement. Il faut tout d'abord savoir que le JavaScript est inséré au sein d'une page HTML, généralement générée à la volée par un script PHP ou ASP qui récupèrent des informations dans une base de données pour les afficher. L'attaque XSS consiste à réussir à placer dans la base de données, à la place d'un pseudonyme

ou de toute autre information destinée à être affichée telle quelle sur le site, un script JavaScript malicieux qui pourra éventuellement récupérer des informations et les envoyer à l'attaquant. Ce script pourra être exécuté dans le navigateur de tout utilisateur qui visite la page attaquée. On peut imaginer récupérer des *cookies*, une adresse IP, ...

Récemment, les plateformes sociales comme Facebook permettent à leurs utilisateurs d'écrire eux-mêmes des applications en JavaScript. La quantité d'information que ces scripts sont susceptibles d'atteindre est énorme, et la question de la confidentialité des données des utilisateurs se pose évidemment.

Les contributions de ce stage sont, d'une part, une sémantique de JavaScript sous forme de système de réécriture de termes, avec laquelle on pourra prouver des propriétés de sécurité, et d'autre part, une analyse statique permettant la preuve automatique d'un théorème permettant l'isolation de différents scripts insérés sur une même page.

Le rapport commencera par établir un état de l'art des analyses de JavaScript existantes en section 1. Nous introduirons ensuite la notion de système de réécriture en section 2. Nous expliquerons ensuite le mécanisme de la sémantique de JavaScript que nous avons écrite (section 3), puis l'analyse que nous avons menée avec cette sémantique (section 4). Nous conclurons enfin en donnant des pistes pour les travaux futurs.

1 État de l'art

Dans cette section, nous présentons diverses approches d'analyse statique ou dynamique du langage JavaScript, visant à assurer des propriétés de sécurité sur les programmes.

1.1 JavaScript : un langage dynamique et flexible

Un langage très dynamique JavaScript est un langage impératif orienté objet, à l'instar de C++ ou Java. À la différence de ces langages, en revanche, il n'existe pas de notion de classe en JavaScript. Les relations entre différents types d'objets sont dictées par des *prototypes*. Les prototypes sont des objets qui représentent un équivalent d'une super-classe (ou classe mère) en programmation objet à classes. Par exemple, lorsque l'on cherche une propriété dans un objet, on cherche d'abord dans l'objet lui-même puis, si la propriété est absente, on va chercher dans son prototype, et ainsi de suite jusqu'à trouver l'objet ou arriver au bout de la chaîne de prototypes. La différence principale par rapport à un langage à classes est la suivante : en Java par exemple, la hiérarchie de classes est statique. En effet, si `A` est une sous-classe de `B`, cela restera ainsi pendant toute l'exécution. En JavaScript, cette hiérarchie est dynamique. On peut, au cours de l'exécution, affecter `C` comme nouvelle valeur du prototype de `A`. De plus, comme évoqué en introduction, tous les objets sont mutables par défaut en JavaScript, en particulier les objets natifs. Ainsi, il est possible de modifier le comportement des objets natifs, en modifiant leur prototype. Par exemple, on peut modifier le comportement de *Object*, dont héritent tous les objets en JavaScript. Changer la valeur de `Object.prototype.toString` modifiera le comportement de tous les objets.

Un langage puissant et flexible JavaScript permet d'accéder à une multitude de composants : on peut réagir aux mouvements et clics de la souris, connaître la géolocalisation de l'utilisateur (si disponible), communiquer avec d'autres pages web ouvertes dans le navigateur ou distantes, accéder à des fichiers sur le système du client, ou encore accéder à la webcam et au microphone de l'utilisateur. Comme évoqué en introduction, JavaScript a accès au DOM, la représentation des pages web du navigateur, et peut donc modifier les pages visitées. JavaScript peut également accéder (lire et écrire) des cookies chez l'utilisateur, ces fichiers qui permettent de garder, entre autres, des informations de session entre deux visites d'un même site. En plus de ces aspects particuliers au domaine du web, JavaScript est également un langage de programmation à part entière et est capable d'exécuter des calculs de la même manière que n'importe quel autre langage.

JavaScript a été conçu pour être simple à apprendre afin de permettre à tout le monde d'écrire des scripts simples pour le web. Ainsi, il est relativement laxiste sur la syntaxe et n'envoie que rarement des erreurs à l'exécution. Dans la plupart des cas où la syntaxe n'est pas respectée, on préfère continuer l'exécution du script *comme si de rien n'était* en donnant une valeur spéciale à l'expression incorrecte. Par exemple, lors de la recherche d'un champ dans un objet, si le champ demandé n'est pas présent dans l'objet, on préfère renvoyer la valeur `undefined` plutôt que d'envoyer une erreur, comme on l'aurait fait, par exemple, en Java ou C++ (au moment de la compilation).

Un langage difficile à analyser Cette flexibilité est appréciable des développeurs JavaScript, mais elle entraîne de nombreuses vulnérabilités, dont le risque est d'autant plus grand que JavaScript est puissant. La plus connue de ces vulnérabilités est le *cross-site scripting* (XSS), dont on a parlé en introduction, qui permet d'injecter du code dans une page qui ne nous appartient pas. Du fait que JavaScript est un langage très dynamique, il est difficile de mener des analyses statiques sur ce genre de programmes. Nous présentons dans la suite de cette section différentes techniques qui visent à analyser JavaScript, qu'il s'agisse d'analyses statiques ou dynamiques.

1.2 Analyse de type

Jensen, Möller et Thiemann [7] proposent TAJS (Type Analysis for JavaScript), un analyseur de type pour JavaScript qui permet d'éviter des erreurs courantes. Par exemple, l'analyseur pourra détecter si on essaie d'utiliser une valeur qui n'est pas de type fonction comme une fonction, ou si on essaie de lire une variable qui n'existe jamais, ou si on essaie de lire une propriété (un champ) dans `null` ou `undefined`. Par exemple, lorsque l'analyseur rencontre un terme du type $f(x_1, \ldots, x_n)$, il s'assurera que f est bien une fonction. Ces trois cas provoquent une erreur à l'exécution et seront donc reportés en priorité par TAJS. L'analyseur est également capable de détecter du code mort (jamais exécuté, ou alors inutile : une écriture dans une variable qui n'est jamais lue par la suite, ou réécrite avant d'être lue).

L'analyse commence par créer une représentation du programme JavaScript sous la forme d'un graphe de flot de contrôle où chaque nœud représente une instruction et

chaque arc représente un chemin possible. Pour l'instruction `if`, par exemple, le nœud contient la condition, puis on a une branche pour le cas où la condition est vraie, et une autre branche pour le cas contraire. Les instructions sont d'une dizaine de types différents, inspirés du codage interne utilisé par certains interpréteurs JavaScript.

Une fois cette représentation obtenue, l'analyse est une interprétation abstraite. L'interprétation abstraite est une théorie de raisonnement sur la sémantique d'un langage dans le but de rassembler des informations sur le programme. L'application principale de l'interprétation abstraite est l'analyse statique. Le principe est le suivant : on commence par définir un domaine *abstrait* des valeurs manipulées (classiquement un domaine moins précis que le domaine original). Ce domaine est organisé dans un treillis. On modélise également l'état abstrait du programme sous la forme d'un treillis. On définit ensuite des *fonctions de transfert* qui permettent de passer d'un état abstrait à un autre, sous l'effet d'une instruction. Le calcul de l'analyse à proprement parler est ensuite une recherche de *point fixe* par application des fonctions de transfert.

Dans le cas de TAJS, le treillis des valeurs est le produit de treillis suivant :

$$Value = Undef \times Null \times Bool \times Num \times String \times \mathcal{P}(L)$$

Tous les sous-éléments sont aussi des treillis, plus simples. Par exemple, $Undef$ (respectivement $Null$) est un treillis à deux élements : `undef` (respectivement `null`) et $\bot$. Le treillis $Bool$ contient quatre élements : `bool` ($\top$), `true`, `false`, et $\bot$. Num est un treillis qui rassemble d'un côté les valeurs infinies, d'un autre les valeurs entières non signées et d'autre part les valeurs signées ou non entières. $String$ sépare les chaînes qui représentent des entiers non signés et les autres chaînes. Cette séparation permettra par la suite de savoir par exemple si une chaîne peut être utilisée comme indice dans un tableau. Enfin, L est l'ensemble des *locations*, c'est-à-dire des pointeurs vers des objets alloués.

Par exemple, la valeur abstraite $(\bot, \bot, true, 42, \bot, \bot)$ désigne donc une valeur qui est soit `true`, soit 42.

L'analyse procède ensuite à l'application des différentes fonctions de transfert. L'analyse rassemble des informations notamment sur les valeurs potentiellement pointées par les références et provoque une erreur si on essaie, par exemple, de lire une propriété de la valeur $(undef, \bot, \bot, \bot, \bot, \bot)$.

1.3 Analyse de flux d'information

Les propriétés de flux d'information permettent d'interdire des flux considérés dangereux. Ces propriétés sont de deux types différents : on peut vouloir garantir que des informations non fiables s'insèrent dans le script (*intégrité*), ou bien que des informations confidentielles soient révélées (*confidentialité*).

1.3.1 Intégrité du programme

La première catégorie concerne les propriétés d'intégrité : on ne veut pas que des informations non fiables s'infiltrent dans des fonctions critiques. Comme on l'a vu en introduction, on peut en JavaScript accéder au DOM et ainsi écrire sur la page. On peut

imaginer une instruction du type `monElement.innerHTML = champ.text` qui va affecter au champ `innerHTML` de l'élément `monElement` le texte contenu dans le champ de texte `champ`. Le champ `innerHTML` représente la structure HTML d'un élément. Ainsi, modifier ce champ modifiera directement la structure interne du document HTML. Il faut donc faire attention à ce à quoi on l'affecte : on peut très bien insérer un script malicieux.

La solution proposée par Guarniera et coll [3] est une analyse de teintes permettant d'éviter ce genre de flux. L'analyse de teintes considère plusieurs types d'élements :

- les sources : ce sont les données d'entrée, typiquement non fiables, dans notre exemple `champ.text` est une source ;
- les puits : ce sont les locations sensibles, dans lesquels on ne veut mettre que des informations de confiance, dans notre exemple `monElement.innerHTML` est un puits ;
- les *sanitizers* : ce sont des fonctions qui permettent de rendre des données issues d'une source invulnérables. Dans notre exemple, la fonction `escape()`, qui permet d'échapper les caractères spéciaux, pour les faire apparaître comme du texte, est un sanitizer.

Les auteurs automatisent le processus d'identification des sources, puits et sanitizers en utilisant des règles du type : "le champ `innerHTML` d'un objet est toujours un puits".

L'analyse consiste ensuite à teinter les variables sources, puis à propager la teinte sur les variables auxquelles on affecte des valeurs teintées.

Une fois cette propagation effectuée, on vérifie si les puits sont teintés : si oui, le programme est potentiellement vulnérable, sinon il est sain.

Soit le programme suivant :

```
y=x;
z=escape(x);
```

Considérons `x` comme étant une variable teintée. La teinte sera propagée à `y` à cause de la première affectation. En revanche, `z` restera non teinté puisque la valeur de `x` ne lui est pas directement affectée, mais on passe par la fonction `escape` qui permet de sécuriser le flux.

Cette analyse permet donc de vérifier que pour tout chemin existant entre une source et un puits, il y a un *sanitizer* qui permet de rendre les données sûres.

1.3.2 Confidentialité des données sensibles

La seconde catégorie de propriétés de flux d'information vise à garantir la confidentialité des données sensibles. Ces propriétés associent à chaque variable un niveau de sécurité : pour simplifier on ne considèrera que les niveaux haut et bas (h ($high$) et l (low)). Cette association est faite au début du programme, puis on vérifie qu'on n'a pas de flux interdits. Les flux interdits sont ceux qui font passer une information de niveau haut dans une variable de niveau bas. On distingue les flux explicites et les flux implicites.

Un flux explicite arrive lorsque l'on affecte directement une variable haute dans une variable basse (`l = h`). Un flux implicite arrive dans des cas comme celui-ci : `if(h){ l = 1 ; } else { l = 0 ; }`. Ici, on n'a pas de flux direct entre les variables l et h,

et pourtant un attaquant connaissant le programme peut deviner la valeur de h en n'observant que la valeur de l.

Hedin et Sabelfeld [5] proposent un système de type dynamique pour implémenter cette analyse pour un sous-ensemble de JavaScript comprenant les objets, les fonctions d'ordre supérieur, les exceptions et l'évaluation de code dynamique. Ils écrivent ensuite une sémantique de leur sous-langage en suivant la spécification officielle, et en y ajoutant des contrôles pour les flux d'information, afin que des informations secrètes (niveau h) ne fuitent pas dans des variables publiques (niveau l).

Ils s'intéressent non seulement aux valeurs dans les variables mais à tous les événements observables. Par exemple, le lancement d'une exception après une condition sur une variable h donne une information sur h à un observateur. Ainsi, cette exception ne sera pas lancée si l'analyse se rend compte qu'une information confidentielle risque d'être révélée.

1.4 Analyse de eval

JavaScript propose une fonction `eval()` qui permet d'exécuter dynamiquement du code. Cette fonction prend en entrée une chaîne de caractères et l'exécute comme un programme JavaScript. Elle est utilisée en pratique comme une forme de réflexivité. Par exemple, pour connaître les versions de Flash installées, on peut exécuter ce code [12] :

```
for(var i = 1; i < 10; i++){
  if(eval("flash"+i+"Installed") == true){ ... }
}
```

Cette utilisation de `eval` n'est pas nécessaire, puisqu'on peut la réécrire en un code du type `window["flash"+i+"Installed"]`. Cependant, elle est présente en pratique.

Une autre utilisation inadéquate d'`eval` est la désérialisation d'objets JSON [1]. JSON est un format d'échange de données entre différents scripts qui permet de transformer un objet quelconque en une chaîne qui le représente. La syntaxe de sérialisation est identique à la syntaxe des objets JavaScript, donc `eval` permet de reconstruire les objets à partir de la chaîne ainsi construite.

```
var x = {a:2};
var x_json = JSON.stringify(x); // x_json contient : {"a":2}
var x_new = eval(x_json); // x_new == x
```

Cela fonctionne, mais la solution adéquate est d'appeler la fonction `JSON.parse()` qui vérifie que l'argument est une chaîne JSON valide, alors que `eval` exécute tout ce qu'on lui donne. Imaginons qu'un attaquant ait réussi à introduire du code malicieux dans la variable `x_json`. En utilisant `eval`, on exécutera le code malicieux, alors qu'en utilisant la fonction JSON adéquate, une erreur expliquant que la chaîne n'est pas une chaîne JSON valide sera renvoyée et on évitera ainsi d'exécuter du code malicieux.

[1] JavaScript Object Notation

Le dernier type d'utilisation recensé [12] permet l'exécution de code arbitraire : mise à jour d'une variable, déclaration de fonction, chargement de librairies. Ces utilisations ne sont pas forcément utiles (on pourrait sans doute se passer d'`eval` dans une partie des cas), mais ces cas d'utilisations sont issus de scripts utilisés sur de vrais sites.

Dans beaucoup de cas (plus d'un tiers [6]), la chaîne passée en argument à `eval` est une constante et est donc connue avant l'exécution. Dans ces cas, on pourrait aisément se passer d'`eval`.

Le danger d'`eval` vient du fait que n'importe quel code puisse être exécuté sans que cela ne soit prévu par le développeur. Il suffit d'injecter du code malicieux dans une variable (par des attaques XSS par exemple) pour que celui-ci soit exécuté.

1.4.1 Collaboration entre une analyse statique et un composant de transformation de code

Jensen, Jonsson et Møller [6] proposent une solution pour éliminer une partie de ces appels inutiles à `eval`.

Ils utilisent une analyse classique standard (TAJS) pour analyser le code qui ne contient pas `eval`, puis lorsqu'un appel à `eval` est rencontré, un composant appelé UNEVALIZER est invoqué pour le transformer en code statique. Ce composant est appelé avec des informations de contexte, par exemple les différentes valeurs possibles pour chacune des variables utilisées dans l'appel à `eval`.

```
if(condition){
  x = "f";
} else {
  x = "g";
}
eval(x+"=2");
```

Dans l'exemple précédent, l'analyseur va inférer qu'après la condition, les deux seules valeurs possibles pour `x` sont les chaînes `"f"` ou `"g"`. L'appel à `eval` va donc être transformé en un code du type `if(x=="f") { f=2; } else { g=2; }`.

Les informations issues de TAJS serviront à l'UNEVALIZER pour déterminer comment transformer le code dynamique (faisant appel à `eval`) en code statique. En pratique, si on a un ensemble fini de valeurs possibles pour une variable représentant un identifiant de variable à laquelle on effectue une affectation, on va *déplier* le code comme dans l'exemple précédent pour tester chacun des cas.

JavaScript propose un mode *strict*, qui permet de changer le comportement de différents aspects de la sémantique y compris celui de la fonction `eval`. L'utilisation du mode strict va modifier le contexte dans lequel on va lire les variables (local ou global) et celui dans lequel on écrit les nouvelles variables. Un deuxième facteur qui modifie le comportement d'`eval` est la façon de l'appeler : directement (`eval(x);`) ou via un alias (`var monEval = eval; monEval(x);`).

Ces différents comportements sont gérés par l'UNEVALIZER, qui applique le comportement adapté grâce aux informations fournies par TAJS.

Le programme est capable de transformer les appels à `eval()` dans plusieurs cas. Dans le cas où la chaîne passée à `eval` est constante, la transformation est assez simple. Il suffit dans la plupart des cas d'*enlever les guillemets* autour de la chaîne et de l'utiliser comme instruction.

Dans les cas où la chaîne n'est pas constante, l'UNEVALIZER peut utiliser les informations de l'analyse statique pour essayer de reconstruire la chaîne par propagation de constantes.

```
var x = "...";
eval("y="+x);
```

Dans l'exemple ci-dessus par exemple, une simple propagation de la valeur de `x` permet de traiter cet appel à `eval` simplement.

L'UNEVALIZER est également capable de traiter les cas où l'argument d'`eval` est une chaîne JSON. Pour savoir si une chaîne est au format JSON, on peut s'appuyer sur une instruction du type `j = JSON.stringify(x)` (c'est une fonction qui transforme un objet en sa représentation JSON). Mais la plupart du temps, les chaînes JSON sont issues de communication avec une base de données distante. On ne sait donc pas quand on fait un appel à une base de données si on récupère une chaîne JSON ou n'importe quelle autre valeur. Pour cela, l'UNEVALIZER s'appuie sur des annotations utilisateur qui explicitent que la chaîne attendue est de type JSON. Pour garder une trace de ces chaînes JSON, le treillis *String* est raffiné pour prendre en compte un type *JSONString*. Ainsi, lorsqu'on appellera `eval` avec en argument une chaîne de type *JSONString*, on pourra transformer l'appel avec la fonction `JSON.parse()`.

Le treillis *String* est à nouveau raffiné pour introduire des types qui représentent les chaînes qui peuvent être des identifiants de champ ou des parties d'identifiant de champ. Ainsi, on pourra analyser les appels du type `eval("x."+prop)` ou `eval("x_"+prop)` en les transformant en `x[prop]` ou `window["x_"+prop]`, ce qu'on n'aurait pas pu faire si `prop` valait `"f*2"` par exemple : on aurait transformé `x.f*2` en `x["f*2"]` !

Ces travaux permettent de transformer environ trois quarts des exemples testés par les auteurs. Les exemples qui ne passent pas font apparaître des appels à `eval` dans des boucles, avec une chaîne représentant un appel de fonction différent à chaque itération. On n'est pas capable dans ce cas là de transformer ce code en code statique.

1.4.2 Utilisation d'analyses statique et dynamique pour transformer le code

D'autres travaux permettent de transformer `eval` en code statique. Meawad, Richards, Morandat et Vitek [10] proposent une solution qui s'appuie sur des informations obtenues à la fois statiquement et dynamiquement.

On commence par naviguer sur la page à tester à travers un proxy nommé EVALORIZER. Le proxy établit un journal des appels à `eval` qui apparaissent sur la page et dans les fichiers inclus. Ce journal contient une liste de triplets (fichier,ligne,appel). Cela

constitue ce que les auteurs appellent le journal statique. Le proxy *instrumente* les appels à `eval` pour enregistrer, lors de chaque appel, les chaînes données en argument à `eval`, ce qui constituera le journal dynamique. Pour le construire, un utilisateur parcourt la page en essayant d'activer tous les appels à `eval` cachés dans du code exécuté après un clic sur un bouton par exemple.

À ce stade, on a donc pour chaque appel à `eval` le nom du fichier et la ligne à laquelle il apparaît, ainsi qu'un échantillon de valeurs passées en paramètre. Pour chaque valeur passée en paramètre, on crée une représentation sous forme d'arbre de syntaxe abstrait (*AST* pour *Abstract Syntax Tree*). Si on a plusieurs valeurs possibles pour un même appel, on fusionne les deux AST avec un nœud de choix (OU) à la racine. Par la suite, on pourra raffiner cette fusion. Par exemple, si les chaînes passées à `eval` sont `window.width` et `window.height`, on va générer l'AST : `OU(.(window,width),.(window,height))`. On peut raffiner ceci par `.(window,OU(width,height))`. On peut aller encore plus loin en généralisant `width` et `height` par n'importe quel identifiant de champ, ce qui est une abstraction suffisante pour transformer ce type d'appel à `eval`.

Une fois cet AST obtenu et généralisé, un composant, le Patcher, est invoqué pour proposer une transformation des `eval`. Dans l'exemple précédent, on pourra remplacer `eval(x)` par un code du type :

```
var regex = /^window\.(ident)$/;
if(match = regex.exec(x)){
  result = window[match[1]];
} else {
  result = eval(x);
}
```

On a défini une expression régulière `regex` à partir de l'AST obtenu précédemment, puis on teste si l'argument `x` est reconnu par l'expression régulière : si oui, on applique la transformation, sinon on appelle `eval` pour conserver le comportement initial du programme. Ce dernier cas ne peut arriver que si on a *manqué* cet appel à `eval` lors de la construction du journal dynamique.

Les auteurs déclarent supprimer 97% des appels à `eval` grâce à cette technique. En effet, l'information supplémentaire obtenue grâce aux journaux dynamique leur permet d'avoir plus d'information que l'UNEVALIZER présenté ci-dessus.

1.4.3 Conclusion sur les analyses d'`eval`

Les analyses présentées dans la partie précédente sont clairement insuffisantes (ne traitant que partiellement `eval` pour transformer les cas reconnaissables et laissant les autres cas plus obscurs inchangés), c'est pourquoi il est nécessaire de trouver une solution qui permette de mener des analyses statiques sur des programmes JavaScript qui contiennent `eval`.

Le problème que pose `eval` est le suivant : alors que les analyses statiques classiques s'intéressent à un programme donné pour vérifier une propriété donnée, avec `eval`, on

doit analyser un ensemble de programmes. En effet, imaginons qu'une analyse d'un programme sans `eval()` nous donne un ensemble E de valeurs possibles pour la variable `x`. Maintenant, comment analyser l'instruction `eval(x)` ?

On ne peut pas envisager de lancer l'analyse précédente pour chaque valeur possible, pour récupérer ensuite l'ensemble des résultats possibles, pour la simple raison que l'ensemble des valeurs possibles peut être très grand, voire infini. Nous verrons dans la suite de ce rapport comment traiter un ensemble de programmes.

1.5 Analyse d'isolation dans des mashups

On trouve souvent sur les pages web plusieurs scripts qui interagissent. Par exemple, un script JavaScript qui affiche de la publicité en lien avec la page visitée inspecte la page dans laquelle il est inséré pour deviner les intérêts du visiteur. Ou encore lors d'une recherche quelconque, on peut trouver un script tiers qui affiche un plan de l'adresse recherchée, avec éventuellement un itinéraire à partir de la localisation de l'utilisateur si disponible. On appelle mashups ces pages où différents scripts interagissent.

Pour insérer différents scripts sur une même page, on a classiquement deux solutions :

- mettre les différents scripts dans des *iframes* différents : les communications entre scripts sont soumises à la *Same Origin Policy* (SOP) : un script ne peut interagir qu'avec des documents venant de la même origine, c'est-à-dire le même sous-domaine, avec le même protocole et le même port de communication ;
- laisser les scripts communiquer librement en les plaçant dans la même *iframe* : on ne peut garantir aucune propriété de sécurité.

Pour faire interagir des scripts afin d'enrichir le contenu des sites web, on doit alors les placer dans une même *iframe*, ce qui pose des problèmes de sécurité. Plusieurs travaux proposent des alternatives plus souples.

1.5.1 Caja : approche au niveau de la compilation

Caja [11] est un sous-ensemble de JavaScript développé par Miller et coll pour éviter ce genre de problème. L'idée principale est d'utiliser un système d'autorisations via le modèle des objets à *capabilities*. Ce modèle stipule qu'un objet A ne peut agir sur un objet B seulement si A possède une référence sur B. L'objet A peut posséder une référence sur B dans quatre cas :

- au début de l'exécution, A connaît déjà B (conditions initiales)
- A a créé B et connaît donc sa référence
- un objet C qui connaît une référence sur B a créé A et a décidé de lui donner sa référence sur B
- un objet C qui connaît une référence sur B et sur A a décidé de donner à A une référence sur B

Ce modèle permet de donner aux différents objets seulement les autorisations dont ils ont besoin et rien de plus. Le système repose sur l'impossibilité de *forger* des références sur les objets.

Concrètement, Caja permet d'écrire des programmes JavaScript en imposant certaines restrictions :

- les identifiants de variable qui finissent par `__` (double *underscore*) sont réservés à Caja
- les identifiants de variable qui finissent par `_` (simple *underscore*) permettent de donner à des propriétés la même politique d'accès que le mot-clé `protected` en Java ou C++ : on n'a accès à cette propriété seulement depuis l'objet lui-même et les objets apparentés (sur la même chaîne de prototypes)
- Caja établit la notion de *module* : ce sont des parties d'application distinctes qui ont chacune leur environnement global. Ainsi, on n'a pas d'environnement global commun à tous les modules.
- Caja permet de *geler* des objets : toute tentative d'altérer un objet gelé (ajouter, modifier ou supprimer des propriétés) lèvera une exception. Les fonctions et prototypes sont automatiquement gelés à leur initialisation, et le programmeur Caja peut geler lui-même des objets.
- Caja interdit l'utilisation de la fonction `eval` ou de la construction `with` (cette dernière construction permet d'utiliser un objet donné comme environnement pour la recherche de variables)

1.5.2 WebJail : approche au niveau du navigateur

Caja étant orienté compilation, il est nécessaire de pouvoir changer le code pour pouvoir utiliser cette technique. Van Acker, De Ryck, Desmet, Piessens et Joosen [13] proposent une autre solution visant à résoudre le même problème d'intégration de différents composants JavaScript dans des mashups. Ainsi, un intégrateur de contenus comme FaceBook qui n'a pas accès au code de l'application mais ne fait que l'insérer sur la page. L'idée de WebJail est de s'assurer qu'une certaine politique de sécurité, définie par l'intégrateur, est respectée, mais de faire cette vérification du côté du client, au sein du navigateur, plutôt que du côté de l'intégrateur. L'idée est encore de ne donner aux différents composants que les objets dont ils ont besoin pour exercer leur fonctionnalité.

L'intégrateur commence par définir sa politique de sécurité. Cela consiste à autoriser ou interdire chacune des actions sensibles recensées. Cette politique s'exprime dans un langage conçu à cet effet et est insérée au moyen d'un attribut `policy` dans la balise HTML `<iframe>`. Les actions sensibles sont divisées en plusieurs catégories : accès au DOM (*Document Object Model*, évoqué en introduction), lecture ou écriture des cookies, communications avec des scripts distants, accès aux données de géolocalisation, etc...La politique est définie comme une *liste blanche* des opérations possibles. Ainsi, une opération non spécifiée dans la politique sera considérée interdite : on peut ainsi aisément autoriser seulement ce qui est nécessaire.

Ensuite, pour insérer un script dans une page, le navigateur va examiner la politique de sécurité renseignée dans l'attribut `policy` du script. Pour toutes les opérations que la politique interdit, WebJail va modifier les fonctions correspondantes de manière à bloquer les opérations frauduleuses.

Ces techniques ont été implémentées dans Mozilla Firefox et testées par exemple sur

la plateforme iGoogle, qui regroupe autant de *gadgets* que l'utilisateur veut en ajouter. Par exemple, ils ont réussi à empêcher l'accès aux données de géolocalisation par le gadget Google Latitude.

1.5.3 Approche par instrumentation du code

Maffeis, Mitchell et Taly [9] proposent une autre solution pour sécuriser les sites qui rassemblent différents scripts d'origine tierce. Ils s'intéressent en particulier à deux propriétés :

- Certaines propriétés des objets ne doivent pas pouvoir être accédées par le code utilisateur. Par exemple, on peut empêcher l'accès à `eval`. Les auteurs généralisent en empêchant l'accès à des propriétés présentes dans une liste noire (*blacklist*).
- Les environnements lexicaux de deux composants distincts doivent être distincts. On ne veut pas que des scripts non sûrs puissent communiquer. Pour ce faire, les auteurs préfixent le nom des variables par un identifiant propre à chaque composant.

Le but est de trouver un sous-ensemble de JavaScript qui satisfait ces deux propriétés.

Le processus commence par une phase d'analyse statique et de filtrage. On commence par refuser tous les programmes qui contiennent un identifiant compris dans la liste noire, `eval`, `constructor` ou `Function` (les deux derniers peuvent être utilisées de la même manière que `eval` pour exécuter du code à partir de chaînes de caractères). Les programmes contenant des identifiants commençant par le caractère `$` sont également rejetés (ces identifiants sont réservés pour une utilisation future : voir paragraphes suivants).

La deuxième phase consiste en la réécriture de certaines constructions pour s'assurer que les vérifications faites statiquement dans le paragraphe précédent soit enforcées dynamiquement. Ainsi, toutes les occurences de `e1[e2]` (accès à un champ en JavaScript) seront réécrites en `e1[idx(e2)]`, où `idx` est une fonction qui évalue son paramètre `e2` et s'assure qu'il ne fait pas partie de la liste noire, qu'il n'est pas égal à `eval`, `Function`, ou `constructor` et qu'il ne commence pas pas `$`. Le code exact de `idx` sera donné dans la section 4.

Une autre étape de réécriture consiste à réécrire les occurences de `this` en `NOGLOBAL(this)`, où `NOGLOBAL(x) = if x == $global then null else x` avec `$global` une référence vers l'objet global. Ce code permet de s'assurer qu'on n'obtiendra pas de référence vers l'objet global via `this`.

Enfin, on réécrit également chaque identifiant de variable `x` en `pid_x`, où `pid` désigne un identifiant propre à chaque programme.

Les auteurs ont prouvé (manuellement) qu'avec ces restrictions, les accès à des propriétés sur liste noire sont impossibles et les composants tiers ne peuvent accéder qu'à une partie des propriétés de l'objet global (celles qui sont accessibles implicitement et admises inoffensives).

2 Systèmes de réécriture

Nous avons besoin d'un formalisme permettant de raisonner sur la sémantique d'un langage, et sur des ensembles de programmes (notamment pour la fonction `eval` mais aussi pour prouver des propriétés sur un ensemble de programmes). Pour modéliser des ensembles potentiellement infinis de programmes, nous utiliserons des automates d'arbre qui permettent de représenter de manière compacte des ensembles infinis. Ces automates ne seront présentés que dans la partie 4.3, puisque nous n'en aurons pas besoin avant. Nous avons choisi de nous intéresser aux systèmes de réécriture qui permettent d'exprimer la sémantique d'un langage et de raisonner sur des automates d'arbres et qui ont été utilisés précédemment pour analyser des programmes écrits en Java [2]. De plus, les systèmes de réécriture permettent d'analyser simplement les états accessibles par des programmes. Dans cette section, nous présentons le formalisme des systèmes de réécriture, dans lequel nous allons écrire notre sémantique de JavaScript.

2.1 Un exemple de système de réécriture

Pour donner l'intuition de ce qu'est un système de réécriture, prenons l'exemple simple des règles arithmétiques de l'addition avec les entiers de Peano. Ces entiers suivent le codage suivant : `z` représente l'entier 0, puis `s(X)` représente l'entier successeur de `X`. Par exemple, le codage de 2 selon Peano est le suivant : `s(s(z))`. Les règles suivantes régissent l'addition :

$$X + z \rightarrow X \quad (1)$$
$$X + s(Y) \rightarrow s(X + Y) \quad (2)$$

Cet exemple est un système de réécriture. À partir d'un *terme* initial (un nombre, ou une opération sur des nombres), on peut *réduire* ce terme en y *appliquant* des règles de réécriture pour arriver à un terme final.

Ainsi, en partant du terme `2+1`, noté $s(s(z)) + s(z)$, on applique les règles (le numéro en indice correspond au numéro de la règle appliquée) :

$$s(s(z)) + s(z) \rightarrow_2 s(s(s(z) + z)) \rightarrow_1 s(s(s(z)))$$

Pour *appliquer une règle*, on doit trouver une règle dont la partie gauche est de la même forme qu'un sous-terme du terme de départ. On peut ensuite substituer à ce sous-terme la partie droite de la règle retenue, en respectant les associations de variables. Dans notre cas, seule la règle 2 peut être appliquée au début, avec $X = s(s(z))$ et $Y = z$. On remplace alors le terme entier par la partie droite : $s(X+Y) = s(s(s(z))+z)$. On procède de cette manière tant qu'on peut trouver un sous-terme et une règle qui correspondent.

2.2 Définition formelle

Cette partie donne une définition formelle de l'intuition donnée ci-dessus. Pour plus de détails, le lecteur pourra se référer à *Term rewriting and all that* [1].

2.2.1 Alphabets et termes

Un alphabet $\mathcal{F}$ est un ensemble fini de symboles de fonction. L'arité d'un symbole est le nombre de paramètres que cette fonction attend. Pour un symbole de fonction d'arité 0, on parlera de constante. On écrira $\mathcal{F}^n$ l'ensemble des symboles de fonction de $\mathcal{F}$ d'arité n.

On considère $\mathcal{X}$ un ensemble de symboles de variables, tel que $\mathcal{F} \cap \mathcal{X} = \emptyset$.

On définit l'ensemble des termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$ l'ensemble défini récursivement par :

- $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, et
- $\forall f \in \mathcal{F}^n, \forall t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

Plus simplement, les termes sont soit des variables, soit des applications de fonction sur des termes.

On définit également l'ensemble des positions d'un terme t, noté $\mathcal{P}os(t)$ par :

- $\mathcal{P}os(t) = \{\epsilon\}$, si $t \in \mathcal{X}$
- $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p | 1 \leq i \leq n \text{ et } p \in \mathcal{P}os(t_i)\}$, si $f \in \mathcal{F}^n$ et $t_1, \ldots, t_n \in \mathcal{X}$

$\{\epsilon\}$ représente la suite d'entiers vide.

On note $t|_p$ le sous-terme présent à la position p dans un terme t. On note $t[s]_p$ le terme obtenu à partir de t en remplaçant le sous-terme à la position p par s.

Par exemple, le terme $s(s(z)) + s(z)$, qu'on notera plutôt $t = +(s(s(z)), s(z))$ a pour ensemble de positions $\mathcal{P}os(t) = \{\epsilon, 1, 1.1, 1.1.1, 2, 2.1\}$. On peut connaître le sous-terme à n'importe quelle position : $t|_{1.1} = s(z), t|_{2.1} = z, t|_\epsilon = t$. On peut remplacer un sous-terme à une position donnée : $t[z]_1 = +(z, s(z))$.

On note $\mathcal{V}ar(t)$ l'ensemble des symboles de variables apparaissant dans le terme t.

Une substitution σ est une fonction $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$. L'application d'une substitution σ à un terme t se note $t\sigma$.

2.2.2 Systèmes de réécriture

Un système de réécriture est un ensemble de règles de la forme $l \to r$, avec $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. On définit une relation de réécriture du système $\mathcal{R}$ notée $\to_\mathcal{R}$: $\forall t, s \in \mathcal{T}(\mathcal{F}, \mathcal{X}), s \to_\mathcal{R} t$ si et seulement si il existe une règle $l \to r$, une position $p \in \mathcal{P}os(t)$, et une substitution σ telles que $l\sigma = s|_p$ et $t = s[r\sigma]_p$.

On note $\to_\mathcal{R}^*$ la fermeture réflexive transitive de $\to_\mathcal{R}$, c'est à dire l'ensemble des termes accessibles un nombre quelconque (éventuellement nul) de réductions.

On dit qu'un terme t est irréductible ou en forme normale si il n'existe aucun terme s tel que $t \to_\mathcal{R}^* s$.

Un ensemble particulièrement intéressant est l'ensemble des descendants, puisque dans le cas d'une analyse statique, il correspond à l'ensemble des états accessibles. L'ensemble des descendants d'un ensemble E de termes par réécriture de $\mathcal{R}$ est défini par :

$$\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) | \exists s \in E, s \to_\mathcal{R}^* t\}$$

Cet ensemble n'est pas toujours calculable. Une raison à cela est que les systèmes de réécriture sont un formalisme Turing-complet. On peut donc exprimer n'importe quel

algorithme dans ce formalisme. Si on pouvait calculer $\mathcal{R}^*(E)$, on aurait exactement l'ensemble des états accessibles de cet algorithme, ce qui est impossible dans le cas général (comme montré par Rice en 1953 : toute propriété non triviale sur la sémantique d'un langage Turing-complet est indécidable).

Dans le cas de l'analyse statique, on s'intéressera à donner une sur-approximation de $\mathcal{R}^*(E)$, qui sera calculable, mais moins précis que l'ensemble lui-même.

Aussi, l'ensemble E de termes initiaux est potentiellement infini. Il nous faut un formalisme qui permette d'exprimer un ensemble de termes infini de manière finie.

2.3 Application à l'analyse statique

Dans le cadre de ce stage, nous nous intéressons à l'analyse statique, c'est-à-dire l'établissement de propriétés en raisonnant sur le code des programmes. Une propriété classique en analyse statique est l'étude des états accessibles d'un programme. Un état de programme est un objet qui représente notamment les valeurs possibles des variables.

Ainsi, nous allons écrire un système de réécriture exprimant la sémantique de JavaScript, c'est-à-dire les changements d'état induits par l'exécution des instructions. Puis nous exprimerons les programmes que l'on souhaite analyser sous forme de termes afin d'obtenir l'ensemble des descendants de ces termes par réécriture de la sémantique. Nous aurons alors l'ensemble des états accessibles par ces programmes.

3 Sémantique de JavaScript

Notre but est de faire de l'analyse statique de JavaScript pour prouver des propriétés de sécurité. L'analyse statique s'appuie sur un modèle du programme (*i.e.* pas le programme lui-même) et raisonne sur ce modèle grâce à des règles de sémantique.

La première étape du stage est donc d'écrire une sémantique de JavaScript.

Plusieurs travaux existants visent à définir une sémantique de JavaScript. En effet, la spécification officielle [2] est relativement verbeuse (250 pages en langage naturel) et ne permet pas de faire des analyses formelles.

Maffeis, Mitchell et Taly [8] ont écrit une sémantique formelle correspondant à la spécification officielle. Pour établir cette sémantique, les auteurs ont interprété la spécification officielle de la manière qui leur semblait la plus naturelle, puis ils ont testé les comportements de différents navigateurs (Mozilla Firefox, Internet Explorer, Safari, Opera et l'interpréteur JavaScript pour Java, Rhino). Cette sémantique a l'avantage de définir le comportement entier de JavaScript, contrairement à d'autres travaux qui ne s'intéressent qu'à un sous-ensemble du langage. Cependant, elle est plutôt volumineuse (70 pages de règles formelles). Il est donc possible de mener des analyses statiques avec cette sémantique, cependant sa formalisation en un langage informatique utilisable pour analyser automatiquement des programmes est difficile [3].

[2] http ://www.ecma-international.org/ecma-262/5.1/

[3] Un doctorant de l'équipe CELTIQUE s'occupe actuellement de formaliser cette sémantique dans le langage de preuves Coq

Guha, Saftoiu et Krishnamurthi [4] ont quant à eux proposé un sous-langage de JavaScript : λ_{JS}. Ils fournissent un compilateur qui transforme un programme JavaScript en programme λ_{JS}, à l'exception de la construction `eval()`. Cette sémantique modélise le langage de manière simple (3 pages). Le fait que la fonction `eval()` ne soit pas gérée ne pose pas de problème particulier, car le compilateur λ_{JS} transforme les appels à `eval()` en un terme qui représente bien l'appel à cette fonction. Ce terme n'est ensuite pas traité par l'interpréteur λ_{JS} fourni par les auteurs. L'idée est ici d'utiliser le compilateur pour obtenir un programme λ_{JS}, mais d'écrire nous même la sémantique qui nous permettra d'analyser le programme donné.

Dans cette section, nous présentons notre sémantique de λ_{JS} en partant d'un langage impératif minimal, puis en ajoutant des fonctionnalités au fur et à mesure. Nous commençons par présenter notre représentation des différentes valeurs manipulées, puis notre représentation de l'état d'un programme sous forme de terme, et nous présenterons enfin les différents aspects de la sémantique.

3.1 Valeurs manipulées

Les valeurs des variables manipulées en λ_{JS} sont de type entier, booléen, chaîne de caractères, objet, référence vers un objet. On compte deux autres valeurs spéciales : *null* et $undefined$.
Les **entiers** sont encapsulés dans un terme appelé *int* et qui prend en paramètre l'entier considéré. Ainsi $int(3)$ représente l'entier 3.
Les **booléens** sont codés par les valeurs *true* et $false$.
Les **chaînes de caractères** sont représentées comme des listes de caractère. On définit chacun des caractères comme des termes constants (d'arité 0), puis on utilise un constructeur *cons* pour créer la liste. La chaîne vide est représentée par la constante *nilstr*. Ainsi la chaîne `"hello"` est représentée par le terme suivant :

$$cons(h, cons(e, cons(l, cons(l, cons(o, nilstr)))))$$

Les **objets** sont également représentés comme des listes. Ici on a besoin d'une liste de couples (identifiant,valeur). On utilise le constructeur *obj* en donnant en premier paramètre un identifiant de champ sous forme de chaîne, en second paramètre la valeur qu'on associe à cet identifiant, et enfin le reste de l'objet. Ainsi un objet `{x :2,y :4}` sera représenté par :

$$obj(str(cons(x, nilstr)), int(2), obj(str(cons(y, nilstr)), int(4), nilobj))$$

Les **références** sont des adresses mémoires qui pointent sur des objets alloués précédemment. Elles sont codées par un terme *loc* qui prend en unique paramètre un entier qui représente une adresse dans le tas (*cf.* la sous-section suivante pour le tas).
Les valeurs **null** et **undefined** sont représentées par des constantes du même nom.

Pour la suite de ce rapport, on écrira les chaînes et les objets comme en JavaScript, c'est-à-dire `"a"` au lieu de $str(cons(a, nilstr))$ et `{x :5}` plutôt que $obj(\text{"}x\text{"}, int(5), nilobj)$. On ommettra également le constructeur *int* autour des entiers.

3.2 État du programme

La sémantique permet d'exprimer les changements de l'état d'un programme après l'exécution d'une instruction. Un état de programme JavaScript est un quadruplet (E, R, H, T) comprenant :

L'environnement E est une représentation des variables et de leurs valeurs. Un environnement est constitué d'un objet représentant les variables déclarées dans le contexte courant, et de son environnement parent (*nilenv* pour l'environnement de plus haut niveau). Ainsi, l'environnement initial est noté *nilenv*, alors que l'environnement après une instruction `var x = 1` est noté *env({x :1},nilenv)*. Ceci est très proche de la manière dont JavaScript modélise les environnements.

La valeur de retour R est mise à jour après chaque instruction pour contenir la valeur retournée par l'exécution de la dernière instruction. Elle peut servir de *tampon* entre deux instructions internes (cf. figure 1). Elle est initialisée à *empty*.

Le tas H est un triplet composé de :
- une clé : c'est l'adresse dans le tas où est stocké l'objet
- un objet : c'est l'objet stocké dans le tas à proprement parler
- le reste du tas : soit d'autres couples (clé,objet), soit *nilheap* pour le tas vide

C'est ici que les objets alloués dynamiquement (avec l'instruction `new A()`) sont stockés.

Le compteur d'allocations T stocke l'adresse dans le tas du prochain objet à allouer. Il est incrémenté après chaque allocation et est initialisé à 0 et est utilisé comme clé pour adresser un objet dans le tas. Dans la pratique, il s'agit d'un entier initialisé à 0.

L'état du programme est alors représenté à l'aide du terme *context* d'arité 4. Un exemple de contexte (ou d'état) est le suivant :

$$\begin{aligned} context(\quad & env(\{"x" : 4\}, nilenv), \\ & true, \\ & heap(1, \{"y" : "f"\}, heap(0, \{"z" : 42\}, nilheap)), \\ & 2) \end{aligned}$$

Ce terme représente un contexte dans lequel on a une variable x ayant pour valeur l'entier 4 et un tas comprenant deux objets correspondant aux objets JavaScript `{y :"f"}` et `{z :42}`. Le compteur est égal à 2 : ce sera l'indice du prochain objet alloué dans le tas, et la valeur de retour est égale à `true`.

3.3 Langage noyau

Nous commençons par présenter la sémantique d'un noyau très simple de langage impératif, qui sera complété au fur et à mesure pour aboutir à λ_{JS}. Le langage considéré au début comprend les instructions suivantes :

- *nil* : l'instruction vide,
- $block(X, Y)$: une séquence d'instructions, on exécutera X puis Y,
- $if(C, X, Y)$: une expression conditionnelle qui s'évaluera en X si C s'évalue en `true`, ou en Y sinon,
- $while(C, X)$: une boucle qui exécutera X tant que C vaut `true`,
- $aff(V, X)$: une affectation de X dans la variable V,
- $varDecl(V)$: une déclaration de la variable V.

Avant de présenter les règles de réécriture, expliquons le principe de fonctionnement de la sémantique. La sémantique code le passage d'un état à un autre sous l'effet d'une instruction. L'exécution d'une instruction est représentée par le terme *exec* qui prend l'instruction à exécuter et le contexte dans lequel on l'exécute. Ainsi, $exec(nil, Context)$ code l'exécution de l'instruction vide (nil) dans le contexte *Context*. L'exécution d'une instruction dans un contexte donne un nouveau contexte. Dans le cas de l'instruction vide, on rend le contexte précédent. Mais dans les autres cas, on peut soit rendre un autre contexte, soit réduire l'exécution de l'instruction demandée à l'exécution d'une autre instruction dans un autre contexte.

Prenons l'exemple de l'instruction $block(X, Y)$ dans le contexte *Context*, ce qui correspond à exécuter X, puis Y en séquence. On commence par exécuter X dans le contexte initial, soit $exec(X, Context)$. Puis, on exécute Y dans le contexte rendu par l'exécution de X, soit $exec(Y, exec(X, Context))$. On est assuré que Y ne s'exécutera que lorsque X sera entièrement exécuté, puisque *exec* attend un contexte de la forme $context(\_, \_, \_, \_)$, ce qui n'arrive que lorsque l'exécution est terminée. L'ordre d'exécution des instructions est bien respecté.

Certaines instructions demandent des opérations intermédiaires : on n'est alors pas capable de rendre directement un contexte ou d'exprimer la succession d'étapes comme dans l'exemple précédent. Pour ces cas-là, on introduit des instructions supplémentaires qui codent chacune une étape intermédiaire et qui utilisent la valeur de retour comme une variable temporaire. Par exemple, l'instruction d'affectation $aff(Var, Val)$ nécessite deux étapes : on commence par évaluer l'expression *Val*, puis on l'affecte à la variable *Var*. Évaluer *Val* s'écrit $exec(Val, Context)$, ce qui se réduit en un contexte $Context_2$ dans lequel la valeur réduite de *Val* est la valeur de retour. Ensuite, on effectue l'affectation en ajoutant l'association de *Val* à *Var* dans l'environnement de $Context_2$. Pour faire le lien entre ces deux étapes, on ajoute un terme (*affDo* dans la figure 1).

La figure 1 présente un langage noyau de λ_{JS} qui permet de montrer le fonctionnement de la sémantique.

Nous avons présenté les deux premières règles dans le paragraphe précédent.

Intéressons-nous aux trois règles définissant le comportement d'un bloc conditionnel *if-then-else*. Le terme $if(C, X, Y)$ correspond à un programme *if* C *then* X *else* Y. Cette exécution se fait en deux temps : d'abord on calcule la condition en évaluant le terme $exec(C, Z)$, ce qui nous rend un contexte ayant pour valeur de retour un booléen `true` ou `false`. On exécute l'une ou l'autre des branches en fonction de ce booléen. Ici, on se sert de la valeur de retour comme *tampon* pour stocker une variable temporaire : le résultat de l'évaluation de la condition.

$$
\begin{array}{lcl}
exec(nil, Z) & \rightarrow & Z \\
exec(block(X, Y), Z) & \rightarrow & exec(Y, exec(X, Z)) \\
 & & \\
exec(if(C, X, Y), Z) & \rightarrow & exec(ifDo(X, Y), exec(C, Z)) \\
exec(ifDo(X, Y), context(E, true, H, T)) & \rightarrow & exec(X, context(E, empty, H, T)) \\
exec(ifDo(X, Y), context(E, false, H, T)) & \rightarrow & exec(Y, context(E, empty, H, T)) \\
 & & \\
exec(while(X, Y), Z) & \rightarrow & exec(if(X, block(Y, while(X, Y)), nil), Z) \\
 & & \\
exec(aff(X, U), Z) & \rightarrow & exec(affDo(X), exec(U, Z)) \\
exec(affDo(X), context(E, U, H, T)) & \rightarrow & context(affEnv(X, U, E), U, H, T)
\end{array}
$$

FIG. 1 – Langage noyau : déclaration de variable, affectation et boucles

La règle suivante transforme simplement une instruction $while(C, X)$ en une condition $if(C, block(X, while(C, X)), nil)$.

Prenons un exemple de programme pour bien comprendre l'exécution de la sémantique. Le programme suivant affecte à z la plus grande des deux valeurs x ou y.

$$if(>(x, y), aff(z, x), aff(z, y))$$

La figure 2 montre les différentes étapes de la réduction de cette expression.

$$
\begin{array}{ll}
 & exec(if(>(x, y), aff(z, x), aff(z, y)), \\
 & \quad context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0)) \\
\rightarrow & exec(ifDo(aff(z, x), aff(z, y)), \\
 & \quad exec(>(x, y), context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0))) \rightarrow \ldots \\
\rightarrow & exec(ifDo(aff(z, x), aff(z, y)), \\
 & \quad exec(>(2, 3), context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0))) \\
\rightarrow & exec(ifDo(aff(z, x), aff(z, y)), \\
 & \quad exec(false, context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0))) \\
\rightarrow & exec(ifDo(aff(z, x), aff(z, y)), \\
 & \quad context(env(\{x : 2, y : 3\}, nilenv), false, nilheap, 0)) \\
\rightarrow & exec(aff(z, y), context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0)) \\
\rightarrow & exec(affDo(z), exec(y, context(env(\{x : 2, y : 3\}, nilenv), empty, nilheap, 0))) \\
\rightarrow & exec(affDo(z), context(env(\{x : 2, y : 3\}, nilenv), 3, nilheap, 0)) \rightarrow \ldots \\
\rightarrow & context(env(\{x : 2, y : 3, z : 3\}, nilenv), 3, nilheap, 0)
\end{array}
$$

FIG. 2 – Exécution pas à pas d'une instruction en λ_{JS}

La première règle de la figure 3 concerne la recherche d'une variable. En effet, une chaîne de caractères $cons(\ldots, \ldots)$ représente un identifiant de variable. Pour que la chaîne soit effectivement reconnue comme une valeur de type chaîne, il faut l'encapsuler dans le constructeur *str*. Le terme *lookup* s'occupe de chercher la variable dans l'environnement, en partant de l'environnement le plus local pour remonter ensuite vers les

$$
\begin{array}{rcl}
exec(cons(X,Z), context(E,Y,H,T)) & \rightarrow & context(E, lookup(cons(X,Z),E),H,T) \\
exec(str(X),Z) & \rightarrow & setRet(str(X),Z) \\
exec(loc(X),Z) & \rightarrow & setRet(loc(X),Z) \\
exec(ref(X),Z) & \rightarrow & exec(refDo, exec(X,Z)) \\
exec(refDo, context(E,X,H,T)) & \rightarrow & context(E, loc(T), heap(T,X,H), T+1) \\
exec(int(X),Z) & \rightarrow & setRet(int(X),Z) \\
exec(obj(X,Y,Z),Z1) & \rightarrow & reduceObj(obj(X,Y,Z), nilobj, Z1) \\
setRet(X, context(E,U,H,T)) & \rightarrow & context(E,X,H,T)
\end{array}
$$

FIG. 3 – Langage noyau : évaluation des valeurs

environnements de plus haut niveau jusqu'à l'environnement global.

Les règles pour évaluer les valeurs sont assez simples : il s'agit de les placer en valeur de retour du contexte. Pour évaluer un objet, on commence par le réduire, de telle sorte que toutes les valeurs de l'objet soient évaluées (c'est le rôle de *reduceObj*, non détaillé ici), puis on le stocke en valeur de retour.

L'allocation d'un objet sur le tas (en JavaScript, `new A()`) est ici notée `ref(X)`. Les règles pour réduire une telle allocation permettent de commencer par réduire d'abord la valeur à stocker dans le tas (`X`), puis de l'ajouter au tas avec comme clé le compteur courant, et enfin d'incrémenter le compteur.

Le terme *setRet* permet d'affecter une nouvelle valeur à la valeur de retour d'un contexte. La dernière règle explique son fonctionnement.

3.4 Objets

Les règles suivantes permettent de définir l'accès à un champ dans un objet : `getfield(X,Y)`, où `X` est un objet et `Y` un identifiant de champ, *i.e.* une chaîne de caractères.
Plusieurs règles sont nécessaire pour réaliser l'ensemble de ces étapes :

1. résolution de l'objet : si X est une variable, on remplace X par sa valeur dans l'environnement
2. résolution des références : si X est une référence, on va chercher la valeur de l'objet référencé dans le tas
3. résolution du nom du champ : si Y est une variable, on remplace Y par sa valeur dans l'environnement
4. parcours de l'objet pour y trouver la propriété souhaitée

Résolution de l'objet

$exec(getfield(X,Y),Z) \rightarrow exec(getfield2(Y), exec(X,Z))$

Lorsqu'on appelle *getfield*, on peut donner différents types de valeur pour l'objet dans lequel on va accéder à un champ : l'objet directement, le nom de la variable qui contient l'objet, ou son adresse dans le tas. La règle précédente permet de placer en valeur de retour du contexte soit l'objet lui-même, soit une référence vers le tas. En

effet, l'effet de *exec* sur un nom de variable sera de placer en valeur de retour la valeur pointée par cette variable. L'effet de *exec* sur un objet ou une adresse est simplement de les placer en valeur de retour.

Résolution des références

$exec(getfield2(Y), context(E, loc(X), H, T)) \rightarrow$
$\quad exec(getfield3(Y), context(E, lookupHeap(X, H), H, T))$
$exec(getfield2(Y), context(E, obj(X, Y1, Z), H, T)) \rightarrow$
$\quad exec(getfield3(Y), context(E, obj(X, Y1, Z), H, T))$
$exec(getfield2(Y), context(E, nilobj, H, T)) \rightarrow$
$\quad exec(getfield3(Y), context(E, nilobj, H, T))$

Ici, la valeur de retour ne peut plus être qu'une référence ou un objet. La première règle pour *getfield2* s'occupe du cas où on a une référence et place en valeur de retour le résultat de la recherche de cette référence dans le tas (*lookupHeap*). Les deux autres cas se contentent de recopier l'objet présent en valeur de retour dans le nouveau contexte. Après cette étape, on est certain d'avoir un objet en valeur de retour ($obj(X, Y, Z)$ ou *nilobj*).

Résolution du nom du champ

$exec(getfield3(Y), context(E, O, H, T)) \rightarrow exec(getfield4(O), exec(Y, context(E, O, H, T)))$

De la même manière on s'occupe du nom du champ, qui peut être contenu dans une variable par exemple. On veut le réduire en une chaîne afin de pouvoir effectuer la recherche elle-même. En exécutant *exec* sur le nom du champ, on obtiendra bien la chaîne voulue quoi qu'il arrive.

Recherche dans l'objet

$exec(getfield4(obj(X, Y1, Z)), context(E, Y, H, T)) \rightarrow$
$\quad exec(getfieldObj(obj(X, Y1, Z), Y), context(E, Y, H, T))$
$exec(getfield4(nilobj), context(E, Y, H, T)) \rightarrow$
$\quad exec(getfieldObj(nilobj, Y), context(E, Y, H, T))$

Une fois que tout est réduit, on délègue à *getfieldObj* la tâche de chercher à proprement parler le champ dans l'objet. Ce terme attend un objet et une chaîne et va chercher dans tout l'objet la propriété voulue. Si elle n'est pas trouvée, on va chercher le champ `prototype` de l'objet et continuer la recherche dans l'objet prototype, jusqu'à arriver au bout de la chaîne de prototypes.

Des traitements similaires sont effectués pour la mise à jour d'un champ d'un objet (*setfield*).

3.5 Fonctions

Les fonctions sont représentées par des termes de la forme : $function(args, body)$ où *args* est une liste d'arguments et *body* est une expression qui correspond au corps de la fonction. La valeur de retour de la fonction est la valeur de retour de la dernière expression évaluée.

L'appel est réalisé au moyen d'un terme *call* qui prend en arguments la fonction à appeler et la liste des paramètres effectifs. Ainsi, le terme $call(F, P)$ correspond à l'appel de la fonction F avec les paramètres P.

$exec(call(X, Z), Context) \rightarrow exec(call2(Z), exec(X, Context))$

$exec(call2(Z), context(E, X, H, T)) \rightarrow exec(callDo(X, Z, nilobj), context(E, X, H, T))$

Lors de l'appel à une fonction, on commence par réduire le premier argument (représentant la fonction) afin de récupérer la fonction à proprement parler et pas une référence dans le tas par exemple.

$exec(callDo(function(args(X, X1), Y), params(Z, Z1), O), context(U1, Y1, H, T)) \rightarrow$
$\quad exec(callDo2(function(args(X, X1), Y), Z1, O), exec(Z, context(U1, Y1, H, T)))$

$exec(callDo2(function(args(X, X1), Y), Z1, O), context(U1, Z, H, T)) \rightarrow$
$\quad exec(callDo(function(X1, Y), Z1, obj(X, Z, O)), context(U1, Z, H, T))$

Les arguments sont représentés par le terme *args* qui code une liste d'identifiants correspondant aux paramètres de la fonction. La liste d'arguments vide est codée par *nilarg*. Les paramètres effectifs de la fonction sont codées de la même manière avec les termes *params* et *nilparam*.

Le but des règles ci-dessus est d'associer chacun des paramètres formels (arguments) à leurs valeurs effectives (paramètres). Pour ce faire, on crée un objet qui prend pour noms de champ les paramètres formels et comme valeurs les paramètres effectifs. Cet objet sera ensuite ajouté à l'environnement pour représenter le contexte de la fonction.

Ainsi, pour l'appel $call(function(args(x, nilarg), body), params(2, nilparam))$, on va créer l'objet $obj(x, 2, nilobj)$ qui servira ensuite d'environnement local.

Les règles ci-dessus font celà en deux temps pour chaque paramètre : on commence par réduire le paramètre effectif en une valeur, puis on effectue l'association dans un second temps.

$exec(callDo(function(nilarg, Y), params(Z, Z1), O), U) \rightarrow err(toomanyparams)$

$exec(callDo(function(args(X, X1), Y), nilparam, O), U) \rightarrow err(toofewparams)$

$exec(callDo(function(nilarg, Y), nilparam, O), context(X1, U, H, T)) \rightarrow$
$\quad callRet(exec(Y, context(env(O, X1), U, H, T)))$

$callRet(context(env(X, X1), U, H, T)) \rightarrow context(X1, U, H, T)$

S'il y a trop ou trop peu de paramètres (deux premières règles ci-dessus), une erreur est envoyée. Lorsqu'il n'y a plus de paramètres, on exécute effectivement le corps de la fonction dans l'environnement nouvellement créé. On encapsule cette exécution de fonction dans un terme *callRet* qui permettra, une fois l'exécution de la fonction terminée, de retourner le contexte appelant, *i.e.* de dépiler l'environnement appelant.

3.6 La fonction eval

La fonction `eval` permet d'exécuter du code à partir d'une chaîne de caractères. Dans notre sémantique, on suppose que la chaîne de caractères passées en argument a déjà été analysée et transformée en un terme encapsulé par *enquote*, ce qui permet simplement d'empêcher l'exécution du code avant l'appel à `eval`. Par exemple, le code JavaScript $eval(\text{"}x = 2\text{"})$ sera disponible sous la forme d'un terme $eval(enquote(aff(x, 2)))$.

$$exec(eval(X), Z) \rightarrow exec(evalRet, exec(X, Z))$$

$$exec(evalRet, context(Y, enquote(X), H, T)) \rightarrow exec(X, context(Y, enquote(X), H, T))$$
$$exec(evalRet, context(Y, X, H, T)) \rightarrow exec(X, context(Y, X, H, T))$$

$$appendProg(enquote(X), enquote(Y)) \rightarrow enquote(block(X, Y))$$

FIG. 4 – La fonction `eval()`

On commence par évaluer l'argument passé à `eval`. S'il s'agit d'une valeur $enquote(X)$, alors on lance l'exécution de X. S'il s'agit d'une valeur différente, on exécute cette valeur (la deuxième règle concernant *evalRet* exprime cette valeur différente : en réalité, on aura une règle par valeur possible, mais dans un souci de concision on ne les représente pas ici).

L'exécution de `eval` dans le contexte local est une simplification de la sémantique de JavaScript. En effet, le contexte dans lequel s'exécute le code dépend de différents paramètres, notamment la façon dont on l'appelle (directement ou *via* un alias) ou encore le contexte dans lequelle on l'appelle (dans le mode strict ou pas). Cependant, l'exécution dans le contexte local nous paraît la plus représentative du comportement général de `eval` : un appel direct sans mode strict.

La dernière règle concerne *appendProg* et permet de concaténer deux programmes pour n'en faire qu'un seul : ainsi, une concaténation de programmes se transforme en un programme constitué d'une séquence de ces deux sous-programmes. On peut alors générer des programmes dynamiquement. Cependant, on fait l'hypothèse qu'on dispose des programmes directement sous forme de termes représentant les programmes, et pas sous forme de chaînes de caractères. On laisse donc une partie d'analyse des chaînes à des travaux ultérieurs.

3.7 Tests

Le développement de la sémantique est assez volumineux : on compte en effet plus de 600 règles de réécriture pour l'intégralité de la sémantique.

Afin d'être sûr que la sémantique n'oublie pas de cas, *i.e.* que tout programme bien formé est bien exécuté par la sémantique, nous avons effectué des tests. Nous avons écrit ou récupéré des scripts de test JavaScript (suite Mozilla). Nous avons traduit ces scripts JavaScript en λ_{JS} grâce au compilateur fourni par ses auteurs. Nous avons adapté

ce compilateur pour qu'il rendre un programme λ_{JS} sous forme de termes directement utilisable par les outils de réécriture.

Nous avons effectué les premiers tests avec MAUDE [4], un outil de réécriture qui applique les règles de réécriture (la sémantique) jusqu'à ce qu'aucune règle ne puisse être appliquée. Grâce à cet outil, nous avons pu détecter des erreurs dans la sémantique (ordre d'évaluation des paramètres, règles manquantes) et y remédier.

Le débogage a pris beaucoup de temps, car les termes générés en λ_{JS} sont très volumineux. En effet, pour un programme quelconque (par exemple `var x = 2 ;`), le terme généré comprend toutes les fonctions et tous les objets natifs de JavaScript, comme les constructeurs des objets de base (`String`, `Object`, `Array`, ...). Le programme généré en λ_{JS} est long d'environ 4000 lignes. Après sélection des objets natifs indispensables et élimination des objets superflus, nous avons un programme d'environ 2000 lignes, qui est un peu plus simple à déboguer.

Pour valider la sémantique, nous avons utilisé les scripts de test fournis avec le compilateur λ_{JS}, puis nous avons comparé les résultats de l'interpréteur λ_{JS} avec les résultats de notre sémantique. Nous ne validons pas la totalité de ces tests, mais nous pensons que notre sémantique est suffisante pour mener des analyses statiques sur JavaScript. En particulier, les mécanismes d'accès aux objets, d'appel de fonction et de chaînes de prototype fonctionnent conformément à λ_{JS}. Des exemples de tests qui échouent mettent en jeu des opérateurs comme le XOR bit à bit : cela ne peut pas fonctionner en λ_{JS} car on n'a pas de représentation binaire des nombres. Les expressions régulières ne sont pas implémentées dans notre sémantique non plus, car notre représentation des chaînes (liste de caractères) ne nous permet pas de le coder efficacement. De plus, cet aspect ne nous semble pas être critique pour la sécurité des scripts.

Une fois la sémantique validée expérimentalement, nous avons décidé de travailler sur les résultats d'un article de Maffeis et coll [9] présenté dans la section 1. La section suivante rappelle leurs résultats et expose l'analyse effectuée dans le cadre du stage.

4 Preuve automatique d'isolation de programmes JavaScript

Cette section présente la preuve effectuée au cours du stage. Cette preuve concerne une analyse statique développée par Maffeis, Mitchell et Taly.

4.1 Objectif de l'analyse

Maffeis, Mitchell et Taly [9] proposent des instrumentations du code JavaScript pour isoler des scripts JavaScript qui s'exécutent sur une même page. Leurs travaux s'inscrivent dans le domaine des plateformes sociales (comme FaceBook par exemple), qui permettent aux utilisateurs de programmer eux-mêmes leurs applications. On se pose alors la question de savoir s'il est sûr de laisser ces scripts interagir avec la page des utilisateurs.

[4] http ://maude.cs.uiuc.edu/

Les auteurs proposent un mécanisme d'isolation de certaines propriétés *blacklistées* des objets et un mécanisme d'isolation entre différents scripts. Ces transformations sont de plusieurs types :

- réécrire le mot clé `this` de manière à rendre `this` si ce n'est pas l'objet global, et rendre `null` sinon ;
- réécrire les identifiants *non-natifs* en les préfixant par le numéro du script auquel il appartient ;
- réécrire les accès à des champs d'objets de manière à empêcher l'accès à un ensemble de propriétés (*blacklistées*).

Ils donnent une preuve manuelle que les transformations empêchent les comportements que l'on souhaite éviter. L'idée est d'utiliser la sémantique exposée dans la section précédente afin de prouver automatiquement ceci.

Nous nous concentrons sur la dernière transformation, *i.e.* tout accès à un champ `e[e2]` est transformé en `e[idx(e2)]`, où `idx` est une fonction permettant de filtrer les noms de champ de la manière suivante :

- si le nom de champ est *blacklisté*, on remplace par la chaîne *bad*
- sinon, on *laisse passer* ce nom de champ

La fonction est décrite dans l'article comme ceci :

$$idx(e) = (\$ = e, \{toString : function()\{return(\$ = String(\$), CHECK\_\$)\}\})$$

$$\begin{aligned} CHECK\_\$ = \quad & (\$BL[\$] \ ? \ "bad" \ : \\ & (\$ == "constructor" \ ? \ "bad" \ : \\ & (\$ == "eval" \ ? \ "bad" \ : \\ & (\$ == "Function" \ ? \ "bad" \ : \\ & (\$[0] == "\$" \ ? \ "bad" \ : \$))))) \end{aligned}$$

Expliquons ce que fait cette fonction. On commence par stocker la valeur de e dans la variable `$`. Ensuite, on rend un objet qui contient une fonction `toString`. Ce qui veut dire que lorsqu'on voudra accéder au champ d'un objet avec ceci pour champ, on va transformer cet objet en chaîne en passant par cette fonction là.

Cette fonction appelle le constructeur `String` sur la variable `$` pour le transformer en chaîne.

Enfin, `CHECK_$` effectue une série de tests sur la variable `$` pour savoir si elle est contenue dans la liste noire (`$BL[$]`), si elle est égale aux chaînes `constructor`, `eval` ou `Function`, ou encore si elle commence par le caractère '$'.

Si l'une de ces conditions est vérifiée, on renvoie `bad`, sinon on renvoie la variable elle-même. On veut vérifier qu'aucun programme instrumenté avec cette fonction *idx* ne peut accéder à une propriété blacklistée.

4.2 Adaptation de la propriété à λ_{JS}

Vu le temps limité et du fait que nous sommes en λ_{JS}, nous avons adapté cette instrumentation à notre cadre de travail. En particulier nous avons modifié la fonction `idx` et l'ensemble de valeurs interdites.

La fonction `idx` est maintenant la suivante :

$$idx(e) = (\$ = e, \$ = String(\$), (\$BL[\$] \ ? \ "bad" \ : \ \$))$$

La première simplification est de rendre directement une chaîne : soit \$ (qui est une chaîne, puisqu'on a appelé le constructeur *String*), soit "*bad*". La fonction est alors plus claire et on n'a pas besoin de transformer l'objet rendu dans la version précédente en chaîne en appelant sa méthode *toString*.

La deuxième simplification est dans la condition, on ne teste plus si la chaîne est égale à *eval* ou *constructor* ou *Function*, on ne garde que la condition de la *blacklist*. En effet, en λ_{JS}, les identifiants *constructor* et *Function* ne correspondent pas à leur valeur en JavaScript et il n'est donc pas pertinent d'en tenir compte dans notre cas. Nous éliminons pour le moment *eval* de notre analyse pour des questions de temps.

Le but de notre analyse est d'exprimer le langage d'entrée de la fonction *idx*, c'est-à-dire l'ensemble des valeurs d'entrée possibles : chaînes, entiers, objets, booléens ; pour ensuite vérifier que les chaînes présentes dans la liste noire (`$BL`) ne sont pas accessibles. Nous souhaitons prouver le théorème suivant :

$$\textbf{Théorème : } \forall v \in Value, idx(v) \notin \$BL$$

Pour exprimer l'ensemble des valeurs possibles $Value$, et l'exécution de l'instruction *idx* sur ces valeurs, nous utilisons un automate d'arbres. Nous présentons cet automate dans la partie suivante, puis nous exposerons la forme de l'automate correspondant à l'analyse de *idx*.

4.3 Automates d'arbres

Les automates d'arbres permettent de représenter de manière compacte et finie un ensemble de termes (ou *arbres*, puisqu'on peut représenter un terme comme un arbre ayant comme racine un symbole de fonction et comme enfants chacun des paramètres sous forme d'arbre). Ces automates sont en fait très semblables aux automates de mots, plus conventionnels, à la seule différence qu'ils permettent d'exprimer des termes.

Prenons par exemple l'automate suivant, avec q_3 comme état final.

$$\begin{array}{rcl} a & \rightarrow & q_1 \\ f(q_1) & \rightarrow & q_1 \\ b & \rightarrow & q_2 \\ g(q_2) & \rightarrow & q_2 \\ h(q_1, q_2) & \rightarrow & q_3 \end{array}$$

On peut vérifier si un terme est *reconnu* par l'automate, par exemple, le terme $h(f(f(a)), g(b))$.

$$\begin{array}{rcl} h(f(f(a)), g(b)) & \rightarrow & h(f(f(q_1)), g(b)) \\ & \rightarrow & h(f(f(q_1)), g(q_2)) \\ & \rightarrow & h(f(f(q_1)), q_2) \\ & \rightarrow & h(f(q_1), q_2) \\ & \rightarrow & h(q_1, q_2) \rightarrow q_3 \in \mathcal{Q}_f \end{array}$$

Ici, le terme se réécrit en un état final de l'automate, il est donc *reconnu par l'automate*.

Formellement, un automate d'arbres est un quadruplet $(\mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta)$. $\mathcal{Q}$ est l'ensemble des états de l'automate, $\mathcal{Q}_f \subset \mathcal{Q}$ est l'ensemble des états finaux. $\mathcal{F}$ est l'alphabet utilisé par l'automate (à mettre en relation avec l'alphabet des systèmes de réécriture). Δ est un ensemble de transitions de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ dans $\mathcal{Q}$. Les transitions sont de la forme $f(q_1, \ldots, q_n) \to q$ (pour $q, q_i \in \mathcal{Q}$).

On note $\to_{\mathcal{A}}$ la relation induite par l'ensemble de transitions Δ de l'automate $\mathcal{A}$. On a ici par exemple $a \to_{\mathcal{A}} q_1, f(a) \to_{\mathcal{A}} q_1, \ldots$. On note $\to^*_{\mathcal{A}}$ la fermeture réflexive transitive de $\to_{\mathcal{A}}$. L'exemple précédent permet de dire par exemple que $h(f(f(a)), g(b)) \to^*_{\mathcal{A}} q_3$.

Les termes t tels que $t \to^*_{\mathcal{A}} q_f, q_f \in \mathcal{Q}_f$ sont dits *acceptés* ou *reconnus*. On définit le *langage reconnu* par un automate comme étant l'ensemble des termes qu'il accepte. Ainsi :

$$\mathcal{L}(\mathcal{A}) = \{t \to^*_{\mathcal{A}} q_f | q_f \in \mathcal{Q}_f\}$$

4.4 Timbuk : un outil de réécriture sur des automates d'arbres

Nous utilisons Timbuk [5] pour effectuer des analyses. Timbuk est un outil de complétion d'automates. On commence par exprimer un système de réécriture (la sémantique λ_{JS}), puis on exprime un automate initial (expliqué ci-dessus).

Timbuk prend l'automate initial, puis complète par réécriture cet automate, c'est-à-dire qu'il ajoute à l'automate tous les états accessibles en une transition. Ce processus est répété jusqu'à ce qu'aucune transition ne puisse être appliquée. La différence principale entre Timbuk et Maude est que l'on travaille sur des automates de termes, soit un ensemble de termes, plutôt que sur un terme donné. Cet outil est donc plus orienté vers la vérification, puisqu'on prouve qu'un terme apparaît (ou pas) pendant la réduction d'un ensemble de termes.

On obtiendra par réécriture un automate d'arbres représentant l'ensemble des valeurs possibles en sortie de `idx`, et il suffira de vérifier que les chaînes *blacklistées* n'apparaissent pas dans cet automate.

4.5 Cadre expérimental

On place dans l'automate initial les fonctions et objets dont on a besoin pour exécuter la fonction `idx`. On met donc dans le tas les objets suivants :

- l'objet global (il contient la variable `$` et les adresses vers `String`, vers la fonction `idx` et vers le tableau `$BL`),
- le constructeur `String`,
- la fonction `idx`,
- et le tableau `$BL` représentant la liste des propriétés interdites.

L'environnement initial contient plusieurs fonctions JavaScript standards telles que `toPrimitive` qui transforment un objet ou une valeur quelconque en une valeur primitive

[5] http ://www.irisa.fr/celtique/genet/timbuk/

(nombre, booléen, chaîne, *undefined*, *null*). Cette fonction standard est utilisée par exemple dans le constructeur de `String`.

Enfin l'instruction que l'on exécute dans ce contexte initial est représentée par l'automate suivant, avec pour état final *qInit* (la définition précise des états `qString`, `qNumber`, ... n'est pas donnée ici) :

$exec(qInstr, qContextInit)$	$\rightarrow$	$qInit$
$idx(qVal)$	$\rightarrow$	$qInstr$
$qString$	$\rightarrow$	$qVal$
$qNumber$	$\rightarrow$	$qVal$
$qBoolean$	$\rightarrow$	$qVal$
$qObject$	$\rightarrow$	$qVal$
$qUndef$	$\rightarrow$	$qVal$
$qNull$	$\rightarrow$	$qVal$
$qStrOk$	$\rightarrow$	$qString$
$qStrBL$	$\rightarrow$	$qString$

Dans le cadre de cette analyse, nous considérons dans un premier temps comme chaîne blacklistée la chaîne vide : *nilstr*. Cela permet de séparer facilement les deux types de chaînes dans l'automate : le chaînes d'au moins un caractère (*qStrOk*) et la chaîne vide interdite (*qStrBL*).

Afin d'accélérer l'analyse, nous n'avons dans *qObject* que des objets susceptibles de se transformer en chaînes, c'est-à-dire des objets qui possèdent une fonction *toString*. Cette simplification ne pose pas de problèmes *a priori*, étant donné que les objets ne possédant pas cette fonction se transforment en une chaîne standard : *[Object object]*, qui est différent de la chaîne *blacklistée* dans notre cas.

4.6 Résultats

Les résultats de cette analyse automatique montrent que la chaîne vide n'est en effet pas accessible à partir de l'automate initial décrit ci-dessus. On a ainsi prouvé le théorème énoncé dans la partie 4.2.

Au cours du développement de cette analyse, nous nous sommes rendus compte d'une anomalie présente dans Timbuk qui avait pour effet de faire une sur-approximation des états accessibles alors que le système de test était conçu pour donner un résultat exact. La description de l'anomalie et notre technique de résolution sont expliqués en annexe A.

On a donc un début de preuve automatique de ce que Mitchell et coll. ont prouvé manuellement.

Pour parvenir à la preuve complète, il faudra exprimer l'automate de l'ensemble des programmes λ_{JS} possibles, appliquer l'instrumentation sur ces programmes, puis montrer que les exécutions de ces programmes ne font pas apparaître d'accès à des champs *blacklistés* d'objets.

Au cours du développement, nous avions en fait commencé par exprimer un automate reconnaissant un ensemble de programmes pour y appliquer les réécritures. Nous nous

sommes finalement concentrés sur la fonction *idx* dans un premier temps afin de régler plus aisément les problèmes auxquels nous avons été confrontés.

On peut donc espérer facilement terminer cette preuve avec notre sémantique. Les étapes envisagées sont les suivantes :

1. exprimer l'ensemble des programmes λ_{JS} complet
2. exprimer en réécriture les transformations du code : pour ce faire on utilisera un terme *rewrite* qui permettra d'ajouter l'instrumentation au niveau des appels à *getfield*, *setfield* et *deletefield*
3. vérifier qu'à aucun moment n'apparaissent des termes du type $getfieldObj(\_, X)$, où X est une chaîne *blacklistée* et *getfieldObj* est le terme qui fait effectivement la recherche dans l'objet (et de la même manière pour *setfieldObj* et *deletefieldObj*).

Cette preuve semble relativement accessible, cependant nous faisons face à quelques difficultés. Notamment, l'automate initial est assez volumineux (plus de 600 états) et le système de réécriture est très volumineux (plus de 600 règles de réécriture), ce qui rend la complétion extrêmement lente. À titre d'exemple, nous avons lancé ce test pendant 16 heures et Timbuk n'a effectué que 6 étapes de complétion pour 16000 transitions ajoutées à l'automate initial. Pour comparer, l'expérience précédente (sur les sorties possibles de la fonction *idx*() prenait un automate un petit peu plus petit et le même système de réécriture mettait 80 minutes pour 775 étapes de complétion pour plus de 37000 transitions ajoutées.

La différence entre ces deux analyses est que l'on veut maintenant exécuter un ensemble de programmes plutôt qu'une fonction sur un ensemble de valeurs (relativement) réduit. Les programmes que l'on souhaite exécuter sont quelconques et Timbuk n'est pas capable de traiter ces programmes de façon exacte. La solution est de définir des approximations, que Timbuk saura interpréter, et qui permettront de pouvoir fusionner des programmes qui seront considérés équivalents par rapport à la propriété que l'on souhaite prouver dans notre cas.

On peut également comparer nos résultats avec ceux de Maude. En effet, ce logiciel permet de faire une recherche de terme dans la réduction. La recherche équivalente à l'analyse que l'on a fait est de chercher le terme $str(nilstr)$, soit la chaîne vide, dans les résultats de la complétion. Les tests ont montré qu'en partant d'un unique terme initial (Maude ne sait pas traiter des ensembles de termes), Maude n'arrive pas à prouver que la chaîne vide n'apparaît pas, alors que Timbuk en est capable en 80 minutes.

Conclusion

Le langage JavaScript, utilisé sur la majorité des sites web actuels, présente un certain nombre d'aspects qui rendent son analyse difficile et donc sa sécurité incertaine. La littérature existante propose différentes approches qui tendent à sécuriser ces scripts de manière à garantir aux utilisateurs un certain niveau de confidentialité lorsqu'ils exécutent ces scripts, mais ces approches s'avèrent insuffisantes pour analyser `eval`.

Ce stage a permis de définir une sémantique de JavaScript avec `eval` sous forme de systèmes de réécriture, ce qui permet de raisonner efficacement sur des ensembles de programmes JavaScript et de prouver des propriétés de sécurité. Le stage a permis de prouver une partie de l'article de Maffeis, Mitchell et Taly [9], et l'autre partie est à présent accessible et fera l'objet de travaux futurs.

Outre cette preuve sur un ensemble de programmes, on peut également analyser un programme donné, sans lui ajouter l'instrumentation définie par Maffeis, pour vérifier que la propriété d'isolation des champs *blacklistés* est déjà vérifiée, c'est-à-dire que le programme a un comportement sûr.

On est aussi capable de prouver certaines propriétés sur des programmes avec `eval`. Avec certaines restrictions ou transformations sur le langage d'entrée, on peut prouver qu'un programme JavaScript avec `eval` se comportera de telle manière et ne provoquera jamais telle action.

L'intérêt du formalisme de réécriture et d'automates d'arbres permet de raisonner sur un ensemble de programmes plutôt que sur un unique programme, comme dans les analyses statiques standards, et de raisonner sur l'ensemble des exécutions possibles de ces programmes, par opposition aux analyses dynamiques. On peut ainsi prouver automatiquement des propriétés de sécurité, plutôt que manuellement [9], et ainsi avoir une plus grande confiance dans les preuves.

Références

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998. 14

[2] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33(3-4) :341–383, October 2004. 14

[3] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, New York, NY, USA, 2011. ACM. 6

[4] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming*, volume 6183 of *LNCS*, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. 17

[5] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society. 7

[6] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44, New York, NY, USA, 2012. ACM. 8

[7] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*, pages 238–255. Springer-Verlag, August 2009. 4

[8] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag. 16

[9] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proceedings of the 14th European Symposium on Research in computer security*, volume 5789 of *LNCS*, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag. 13, 25, 31

[10] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone ! : semi-automated removal of eval from javascript programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 607–620, New York, NY, USA, 2012. ACM. 9

[11] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - Safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008. 11

[12] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. 7, 8

[13] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail : least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM. 12

A Découverte d'une anomalie dans TIMBUK

Au cours des tests, nous avons dû faire face à une anomalie de TIMBUK qui fusionnait des contextes alors que nous ne le voulions pas. Nous expliquons dans un premier temps un aspect particulier du fonctionnement de TIMBUK avant d'expliquer le bug rencontré.

TIMBUK commence par prendre l'automate initial et chercher toutes les règles qui peuvent s'appliquer sur chacun des états de cet automate. Pour chacun des termes atteints, TIMBUK cherche dans l'automate si ce terme est déjà reconnu :

- dans le cas où le terme n'est pas déjà reconnu dans l'automate, TIMBUK crée un nouvel état pour ce terme,
- dans le cas contraire, l'ancien état est utilisé à la place.

Maintenant, au cours de la complétion, on crée un terme reconnu par un état q_0 du type $exec(if(\$BL[\$], "bad", \$), qContext)$, avec $qContext$ un état qui reconnaît deux contextes distincts :

- un contexte dans lequel la variable \$ est égale à la chaîne vide *blacklistée*,
- un contexte dans lequel la variable \$ reconnaît toutes les autres chaînes.

Rappelons les règles définissant l'exécution d'une conditionnelle :

$$\begin{aligned} exec(if(C,X,Y),Z) &\rightarrow exec(ifDo(X,Y), exec(C,Z)) \\ exec(ifDo(X,Y), context(X1, true, H, T)) &\rightarrow exec(X, context(X1, true, H, T)) \\ exec(ifDo(X,Y), context(X1, false, H, T)) &\rightarrow exec(Y, context(X1, false, H, T)) \end{aligned}$$

On va alors générer ces deux termes dans q_0 :

$exec(ifDo("bad", \$), context(\_, true, \_, \_))$
$exec(ifDo("bad", \$), context(\_, false, \_, \_))$

Le premier de ces deux termes va se réécrire en $exec("bad", context(\_, true, \_, \_))$. Cependant, pour introduire ce nouveau terme dans l'automate, on va utiliser l'état $qContext$ pour reconnaître le contexte. Le problème est que cet état reconnaît également un autre contexte, pour lequel on ne devrait pas exécuter cette instruction ("*bad*"), mais \$. Le même phénomène se produit pour l'autre contexte, de façon symétrique. Ainsi, on produit ces quatres termes au cours de la complétion, au lieu de 2 normalement (notés en gras) :

exec("bad",context(_,true,_,_))
$exec("bad", context(\_, false, \_, \_))$
$exec(\$, context(\_, true, \_, \_))$
exec($,context(_,false,_,_))

Pour résoudre ce problème, nous avons modifié les règles qui définissent l'évaluation du terme $ifDo$ de la manière suivante :

$$\begin{aligned} exec(ifDo(X,Y), context(E, true, H, T)) &\rightarrow exec(X, context(E, \mathbf{empty}, H, T)) \\ exec(ifDo(X,Y), context(E, false, H, T)) &\rightarrow exec(Y, context(E, \mathbf{empty}, H, T)) \end{aligned}$$

Nous remplaçons les valeurs `true` et `false` par le terme `empty` qui indique une absence de valeur de retour. Cela n'affecte pas la validité de la sémantique puisque cette valeur est issue de l'évaluation de la condition et nous l'avons déjà utilisée. En revanche, celà permet de ne pas réutiliser l'état existant et ainsi ajouter des termes indésirables.