UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉTUDE DES LIMITATIONS DES APPROCHES EXISTANTES D'ANALYSE STATIQUE DE PROGRAMMES JAVASCRIPT

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

OLIVIER ARTEAU

DÉCEMBRE 2025

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

Plusieurs projets logiciels JavaScript modernes ont maintenant un ordre de grandeur dans les milliers ou les millions de lignes de code. Pour arriver à identifier efficacement des vulnérabilités logicielles dans ces projets ou dans un large volume de ceux-ci, il est nécessaire d'avoir un certain niveau d'automatisation. À ce titre, une très grande variété d'approches existe. On classifie généralement ces approches en trois catégories : les analyses dynamiques où on infère des vulnérabilités en fonction du comportement du projet à l'exécution, les analyses statiques où on infère des vulnérabilités en regardant le code du projet sans l'exécuter et les analyses hybrides où l'on combine les deux précédentes.

Pour faire de la recherche de vulnérabilités logicielles à grande échelle, les approches d'analyse dynamique ont le défaut d'être difficiles à mettre en œuvre, car les projets ont souvent des particularités d'exécution qu'il faut prendre en compte. Faire des ajustements pour chaque projet est généralement ardu et exigeant en temps. C'est pourquoi les approches d'analyse statique sont souvent privilégiées pour faire de la recherche de vulnérabilités logicielles à grande échelle. Comme un des objectifs personnels de ce projet de maîtrise est d'améliorer la recherche de vulnérabilités logicielles à grande échelle, c'est ce segment d'approches qui sera approfondi.

L'analyse statique elle-même a plusieurs sous-catégories d'approches. Celle qui nous intéresse le plus est celle où on modélise le code du projet sous forme de graphe et où l'on infère des vulnérabilités en regardant s'il existe des liens entre des puits et des sources à partir de ce graphe. Une des raisons pour laquelle

cette approche est intéressante est qu'elle permet de représenter une large proportion des catégories de vulnérabilités. Un exemple est présenté dans l'article de recherche "Mining NodeJS Vulnerabilities via Object Dependency Graph and Query" (Li *et al.*, 2022). Cet article s'intéresse à la modélisation de vulnérabilités les plus fréquemment présentes dans les bibliothèques NPM. Il note au passage qu'une approche d'analyse d'AST et de graphe de flux de contrôle est capable de représenter 13 des 16 catégories de vulnérabilités présentes dans les bibliothèques NPM.

Le projet de recherche va se concentrer uniquement sur le langage JavaScript. Ce langage a principalement été choisi pour continuer à approfondir mes recherches précédemment publiées (Arteau, 2018). Un autre point d'intérêt de ce langage est sa difficulté d'analyse. L'absence de typage des variables, la construction dynamique des classes et les noms dynamiques de propriétés sont tous des caractéristiques du langage JavaScript qui rendent difficile l'analyse statique de code. Ces caractéristiques font en sorte qu'il est difficile d'établir des contraintes sur ce qu'une variable peut contenir pour les cas non triviaux. L'absence de typage rend difficile la tâche d'établir le type d'une variable dans les cas non triviaux. La construction dynamique des classes fait en sorte qu'il est difficile de contraindre les types qui peuvent exister, puisqu'ils peuvent être créés et modifiés dynamiquement. Les accès dynamiques aux propriétés font en sorte qu'il n'est pas toujours trivial d'établir le nom des propriétés accédées.

Le contexte dans lequel les approches d'analyse statiques sont utilisées pour faire de l'identification de vulnérabilités peut être décrit comme suit.

1. On a tout d'abord des approches qui produisent une représentation du code et/ou transforment le code que l'on veut analyser.

2. Ensuite, il y a des approches qui analysent la représentation du code pour

identifier où les caractéristiques que l'on veut identifier sont présentes.

3. Ce qui est identifié doit ensuite être trié par un processus manuel et/ou automatisé pour vérifier si l'identification correspond effectivement à une vulnérabilité.

Dans le contexte de ce mémoire de maîtrise, on va s'intéresser uniquement aux éléments 1 et 2. 3 est exclu principalement parce qu'il est difficile d'appliquer de façon uniforme des algorithmes de tri manuel ou automatisés pour toutes les approches de 2. On risque donc d'avoir des résultats dont il est difficile d'évaluer si le résultat est attribuable aux approches 1 et 2 ou aux approches 3. Se limiter seulement à 1 et 2 permet de cibler quelque chose de plus précis et plus facilement analysable.

L'état de l'art dans ce domaine ne permet pas forcément de comprendre les limitations des approches existantes, les potentiels de recherche restants et de comparer les approches présentées dans les articles. Cela découle principalement du fait que pratiquement chaque article de recherche utilise des jeux de tests différents. Au final, peu de résultats publiés sont comparables. La majorité des articles regardent uniquement la performance de leur approche sur des cas de leur choix et analysent peu les cas où leur approche a de mauvaises performances. On voit par exemple des phrases comme « We have excluded the package esprima from the recall measurements because it has been bundled using webpack. » (Nielsen *et al.*, 2021) ou « Excluding those that timed out or crashed in our data collection process » (Zhang et Meng, 2020) sont présents dans des articles publiés dans des revues de l'ACM et des conférences. C'est pourquoi le projet de maîtrise porte sur l'amélioration de la compréhension de ces approches par la construction d'un banc de test et de l'analyse des différents aspects mesurés sur chaque approche et outil sélectionné. Ceci permettra d'établir des fondations plus solides pour la recherche dans le domaine de l'analyse statique de code JavaScript.

0.1 Concepts principaux

Pour motiver ce projet de recherche, plusieurs concepts centraux sont importants à présenter. Ce sont tous des concepts largement utilisés dans la littérature.

0.1.1 Point fixe

Un point fixe d'une fonction est mathématiquement défini comme étant une valeur qui est appliquée à cette fonction et qui retourne cette même valeur. Dans le contexte de l'analyse statique de code, un point fixe représente généralement un état (la valeur) pour lequel un algorithme d'analyse statique (la fonction) n'est plus en mesure d'ajouter de l'information supplémentaire (retourne la même valeur). Le concept de point fixe est généralement utilisé pour désigner le point de terminaison d'une analyse.

0.1.2 Supremum ($\top$)

$\top$ est une valeur particulière qui, si elle existe, dénote la plus grande valeur dans un ensemble partiellement ordonné. Cette valeur particulière a comme caractéristique que toutes les autres valeurs dans l'ensemble y sont inférieures. Dans le contexte de l'analyse statique de code, on utilise généralement $\top$ pour désigner que n'importe quelle valeur de l'ensemble est possible. Par exemple, on peut désigner qu'une variable d'un programme peut contenir les chaînes de caractères "foo" ou "bar" en utilisant l'ensemble { "foo", "bar" }. Si on veut désigner que n'importe quelle chaîne de caractères est possible, on utilise la valeur $\top$.

0.1.3 Infimum ($\perp$)

$\perp$ est une valeur particulière qui, si elle existe, dénote la plus petite valeur dans un ensemble partiellement ordonné. Cette valeur particulière a comme caractéristique que toutes les autres valeurs dans l'ensemble sont supérieures. Dans le contexte de l'analyse statique de code, on utilise généralement $\perp$ pour désigner qu'aucune valeur n'est possible. Le principal contexte d'utilisation de $\perp$ est lorsqu'il n'existe aucun chemin qui se rend à une variable. Quand il n'existe aucun chemin qui se rend à une variable, on considère que cette variable ne peut jamais avoir de valeur. On emploie aussi l'expression *code mort*.

0.1.4 Élargissement

Le concept d'élargissement a été introduit pour l'analyse de programme par Cousot et Cousot (1977). L'élargissement se définit principalement comme une fonction qui produit un ensemble de dimension égale ou supérieure à celui qu'il a reçu en argument. Dans le contexte de l'analyse statique de code, on utilise généralement ce type de fonction pour éviter d'avoir à considérer individuellement un grand nombre de valeurs possibles distinctes. Dans ce contexte, l'élargissement substitue un grand nombre de valeurs possibles par une valeur qui représente quelque chose de plus large ou égal que cet ensemble. On peut, par exemple, passer d'un ensemble { 1, 3, 5, 6, 10 } à un ensemble qui est défini comme tous les nombres entre 1 et 10. L'utilisation de ce type d'approche facilite et/ou accélère souvent l'obtention d'un point fixe.

0.1.5 Rétrécissement

Le concept de rétrécissement a été introduit pour l'analyse de programme par Cousot et Cousot (1977). Le rétrécissement se définit principalement comme une technique qui consiste à appliquer la fonction d'analyse à un point fixe obtenu après élargissement systématique pour se rapprocher du point fixe sans élargissement. En pratique, le rétrécissement sert principalement à contrebalancer la surapproximation causée par l'élargissement.

0.1.6 Graphe d'appel

Un graphe d'appel est un graphe orienté qui représente la relation entre l'entité qui appelle et celle qui est appelée pour les fonctions d'un programme. Ainsi, si dans un programme la fonction "a" appelle la fonction "b", le graphe d'appel de ce programme comprendra un arc de "a" vers "b".

0.1.7 Graphe de flux de contrôle

Un graphe de flux de contrôle (Control Flow Graph - CFG) désigne un graphe orienté qui représente l'ordre d'exécution des instructions. Les nœuds sont les instructions et les arcs vont d'une instruction aux suivantes.

0.1.8 Analyse dataflow

L'analyse dataflow est une approche pour établir des propriétés d'un programme qui nécessite un graphe du flux de contrôle. On utilise principalement ce type d'analyse pour approximer les valeurs possibles des variables et expressions d'un programme.

0.1.9 Analyse d'atteignabilité

L'analyse d'atteignabilité en analyse de programme est une approche pour déterminer si un segment de code ou une valeur peut atteindre un autre segment de code via des appels de fonction et/ou des manipulations de données. On désigne qu'un segment de code est atteignable par un autre segment de code ou une valeur s'il existe au moins un chemin dans le programme pour lequel cette relation est vraie.

0.1.10 Sensibilité au contexte

La sensibilité au contexte est une caractéristique d'une approche d'analyse de programme. Cette caractéristique définit que, lors de l'analyse, les fonctions sont considérées en prenant en compte le contexte de leur invocation. Cette caractéristique est présente, par exemple, dans ce qu'on appelle k-CFA (Vitek *et al.*, 1992; Shivers, 1991). k-CFA fait ici référence à une analyse de flux dans laquelle on conserve k éléments de la chaîne d'appel (contexte).

0.1.11 Analyse symbolique

L'analyse symbolique désigne une analyse qui utilise des valeurs symboliques pour représenter les valeurs et les états d'un programme. Ces valeurs symboliques sont composées de symboles et de valeurs concrètes. Les symboles représentent des valeurs dans un système de contraintes. Par exemple, une analyse symbolique va déterminer que pour le code `var a = b + 3`, la valeur abstraite de `a` est la somme du symbole `b` et de la valeur concrète 3.

0.1.12 Complétude

La complétude, en analyse de code, est une caractéristique qui désigne qu'une approche ou un algorithme retourne toujours des résultats vrais. Il est bon de noter qu'une approche ou un algorithme complet ne retournent pas nécessairement tous les résultats vrais. Une approche ou un algorithme complet préfère avoir des faux négatifs que des faux positifs.

0.1.13 Sûreté

La sûreté, en analyse de code, est une caractéristique qui désigne qu'une approche ou un algorithme retourne l'ensemble des résultats vrais. Il est bon de noter qu'une approche ou un algorithme sûr ne retournent pas nécessairement uniquement des résultats vrais. Une approche ou un algorithme sûr préfère avoir des faux positifs que des faux négatifs.

0.2 Plan du mémoire

Dans ce mémoire, les objectifs de recherche expliquent les questions précises sur lesquelles ce projet de recherche va se concentrer. La section "état de l'art" expose ce qui s'est déjà fait et écrit par rapport au projet de recherche. La section "méthodologie" présente comment les résultats des expérimentations sont obtenus. La section "expérimentations" présente le détail de ce qui a été exécuté et les résultats bruts qui ont été obtenus. La section "analyse des résultats et discussion" présente les analyses et constats qui ont pu être faits par rapport aux résultats des expérimentations. La conclusion résume les résultats et les contributions de ce projet de recherche.

0.3 Objectifs de recherche

Pour comprendre l'objectif de ce travail de recherche, on se doit de revenir sur un principe fondamental de l'analyse de code. L'extraction de propriétés complexes de programme arbitraire est un problème indécidable (Rice, 1953). Ceci découle du fait que le problème de l'arrêt est indécidable. Pour faire de l'analyse statique de code, on doit donc utiliser des approches qui sont approximatives. Trouver une approche "idéale" n'est donc pas une question de trouver l'approche qui peut tout trouver, mais davantage une question de trouver une approche qui fait les bons compromis entre avoir de la précision et être performant. L'idée de l'analyse que l'on veut faire est donc d'identifier là où les approches existantes sont précises ou imprécises et là où elles sont performantes ou non. Le tout doit servir à obtenir une meilleure compréhension de l'état de l'art et des opportunités d'amélioration. En termes plus précis, on essaie de répondre aux questions de recherche suivantes.

— Q1 - Quelle précision peut-on s'attendre pour de l'analyse sur des programmes JavaScript réels avec les approches existantes ?
— Q2 - Quelles structures de programme sont difficiles à analyser ?
— Q3 - Pourquoi certaines approches sont précises ou performantes ? Pourquoi certaines approches sont imprécises ou non performantes ?

On veut aussi, à travers ce projet de recherche, regarder certains aspects spécifiques d'approches décrites dans la littérature.

— Q4 - Est-ce que la précision obtenue dans l'article Feldthaus *et al.* (2013) reste équivalente pour les très gros projets ?
— Q5 - Est-ce que l'analyse précise de boucle présentée dans Park et Ryu (2015) amène un gain potentiel de précision pour l'analyse des très gros projets ?

0.3.1 Q1 - Quelle précision peut-on s'attendre pour de l'analyse sur des programmes réels avec les approches existantes ?

Cette question vise essentiellement à connaître et établir une ligne de base de l'état de l'art. Actuellement, quand on regarde les articles publiés qui discutent d'analyse statique de code JavaScript, il est difficile d'établir si l'approche proposée est globalement meilleur, meilleur uniquement dans certains cas ou pire que ce qui est déjà existant. L'établissement d'une base de référence permettrait à la fois de mieux comprendre l'état de l'art et aider à mieux comprendre ce qui est publié dans le futur.

0.3.2 Q2 - Pour quelles structures de programmes les approches sont précises ou imprécises et performantes ou non ?

Cette question vise à identifier des cas où les approches proposées atteignent leur limite et ne sont plus capables d'analyser un programme soit pour des raisons de performance ou de précision.

0.3.3 Q3 - Pourquoi certaines approches sont précises ou performantes ? Pourquoi certaines approches sont imprécises ou non performantes ?

Ces deux questions cherchent à comprendre ce qui fait en sorte qu'une approche proposée analyse bien ou non des programmes donnés. Contrairement à la question Q2 où on cherche à établir les limites des approches, ces deux questions cherchent à établir pourquoi ces limites sont présentes là où elles sont.

0.3.4 Q4 - Est-ce que la précision obtenue dans l'article Feldthaus *et al.* (2013) reste équivalente pour les très gros projets ?

Cette question est intéressante, parce que certaines caractéristiques du code peuvent influencer la précision d'approche décrite dans Feldthaus *et al.* (2013). Il est intéressant de regarder pour de plus gros projets si la précision décrite dans l'article original est toujours présente et si oui dans quelle mesure.

0.3.5 Q5 - Est-ce que l'analyse précise de boucle présentée dans Park et Ryu (2015) amène un gain potentiel de précision pour l'analyse des très gros projets ?

Le principal article de recherche qui discute d'analyse précise de boucle (Park et Ryu, 2015) n'analyse pas son approche sur des projets de dimensions très grandes. L'idée de cette question de recherche est surtout d'évaluer l'importance d'utiliser ce type d'analyse sur de grands projets. Est-ce que l'on obtient un gain de précision ? Si oui, dans quelle mesure ?

CHAPITRE I

ÉTAT DE L'ART

1.1 Sélection des articles

La sélection des articles de recherche pour établir l'état de l'art s'est faite selon les critères suivants. Pour être inclus, l'article doit présenter les caractéristiques suivantes.

- L'article doit traiter d'analyse de programmes JavaScript.
- L'approche d'analyse doit être statique. Elle ne doit pas contenir d'éléments découlant de l'exécution du programme pour extraire les caractéristiques qu'elle veut obtenir.

Les articles ont été présélectionnés en faisant une recherche avec les mots clés "JavaScript" et "static" à partir des moteurs de recherche de IEEExplore et ACM. Pour étendre la recherche, les articles cités par les articles trouvés ont été inclus dans la présélection.

Le tableau 1.2 présente le sommaire des principales caractéristiques (type d'éléments analysés, nombre d'éléments sélectionnés et ce qui est mesuré) des articles qui ont été trouvés pour le survol de l'état de l'art.

Article	Type	Nombre	Mesure
(Lee et Son, 2023)	S	100,000	P, R, A
(Wei et Ryder, 2015)	L, B, E	28	T, VP
(Seifert *et al.*, 2022)	-	-	-
(Parameshwaran *et al.*, 2015)	S, B	1165	T, FP, VP
(Taly *et al.*, 2011)	L	3	T, VP, FP
(Park *et al.*, 2016)	S	30	T, VP, FP
(Andreasen et Møller, 2014)	L	83	T, A
(Feldthaus *et al.*, 2013)	L	10	R, P
(Fass *et al.*, 2021)	E	19,577	A
(Julian *et al.*, 2012)	L	5	T, A
(Li *et al.*, 2021)	B, L	48,214	T, VP, FP, FN
(Guarnieri et Livshits, 2009)	E	8,379	T, VP, FP
(Meawad *et al.*, 2012)	S	100	T, A
(Zhang et Meng, 2020)	S	100,000	T, A
(Guarnieri et Livshits, 2010)	O, M	Non spécifié	T
(Younang et Lu, 2015)	B, E, O	19	T, A
(Nielsen *et al.*, 2019)	S	11	T, VP, FP
(Kashyap *et al.*, 2014)	E, M, O, B	28	T, P
(Jang et Choe, 2009)	O, L	13	T, A
(Madsen *et al.*, 2013)	E	25	T, VP
(Ko *et al.*, 2015)	S, L, O	15	P, C
(Jensen *et al.*, 2012)	S	22	A
(Guarnieri *et al.*, 2011)	S, B	209	VP
(Park et Ryu, 2015)	S, L, O	145	T, VP
(Shiyi et Ryder, 2014)	S	12	T, A

(Chugh *et al.*, 2009)	S	100	T, A
(Madsen *et al.*, 2015)	O	18	T, A
(Welearegai *et al.*, 2019)	L, O	15	T, P, C
(Park *et al.*, 2015)	L, S	15	T, C
(Andreasen *et al.*, 2017)	-	-	-
(Anderson *et al.*, 2005)	-	-	-
(Anders *et al.*, 2009)	B, L	157	VN, FN, FP, VP
(Kashyap *et al.*, 2013)	B, L, O	21	T, A
(Nielsen et Møller, 2020)	B, L	13	T, VP
(Bandhakavi *et al.*, 2011)	E	1827	A

TABLEAU 1.2 – Sommaire des articles trouvés. **S** - Site web **L** - Bibliothèques **O** - Code source ouvert **B** - Banc de test **E** - Extension **M** - Code créé à la main ou généré **T** - Temps d'exécution **VP** - Vrai positif **VN** - Vrai négatif **FP** - Faux positif **FN** - Faux négatif **R** - Rappel **P** - Précision **F** - Score F **C** - Couverture de code (quantité de code atteint par l'analyse) **A** - Autres éléments spécifiques à l'article

1.2 Méthodologie d'évaluation et comparaison de moteurs d'analyse statique de code JavaScript

Dans les articles publiés, le seul article qui a pu être trouvé qui présente une approche méthodologique pour comparer des outils d'analyse statique de code est (Antal *et al.*, 2023). Cet article fait uniquement le comparatif des outils sur des petits fichiers JavaScript. L'analyse de plus gros projets comporte son lot de difficulté au niveau de la méthodologie. L'approche de l'article n'est pas transposable telle quelle pour la recherche que l'on veut faire, étant donné que la méthodologie

utilisée est uniquement adaptée à de petits projets. De plus les objectifs de cet article sont légèrement différents de ce que l'on veut étudier dans ce projet de recherche. L'article fait avant tout une comparaison quantitative de la précision des graphes d'appel générés, alors que, dans ce projet de recherche, on cherche à comprendre les limitations des approches existantes (évaluation qualitative).

Ce que l'on voit en dehors de cet article, c'est principalement de la méthodologie qui est faite sur mesure pour chaque article. Afin de mettre de l'avant que leur approche est pertinente et intéressante, ces articles évaluent un ou plusieurs aspects de leur approche. Dans les articles trouvés, ce qui est évalué est très dépendant du sujet traité. Par exemple, si on traite d'analyse symbolique de code, on va souvent évaluer la couverture du code, mais si on traite de construction de graphes d'appel, on va souvent évaluer la précision de ce graphe et la performance de sa construction.

Ce qui se dégage des articles trouvés est que les aspects suivants sont généralement mesurés :

- Temps d'exécution réels (en secondes ou minutes)
- Support du langage (est-ce que toute la syntaxe, toutes les versions et toutes les fonctionnalités du langage sont supportées ?)
- Quantité ou taux de vrai positif, faux positif, vrai négatif, faux négatif.
- Ratio de précision (précision, rappel et F)
- Couverture du code (est-ce que toutes les lignes du code ont été analysées ou atteintes ?)

Une fois les métriques établies, l'autre aspect de l'évaluation de l'approche proposée est la sélection des éléments de tests. Ce qu'on remarque dans les articles trouvés est que pratiquement chaque article choisit un jeu de code unique à son article et que l'utilisation d'éléments comparables d'un article à l'autre est rarement présente. On arrive donc au constat que, dans la très grande majorité des cas, les

résultats présentés dans les articles scientifiques qui traitent de l'analyse statique de code JavaScript ne sont pas comparables à d'autres articles qui traitent du même sujet ou d'un sujet connexe. On note aussi une absence fréquente de publication du code utilisé pour l'évaluation. Ceci arrive principalement parce que les articles font référence à des éléments qui varient dans le temps (ex. : le code JavaScript contenu dans la page principale du top 100 Alexa [1]). On arrive donc à un deuxième constat que la plupart des résultats publiés dans les articles ne sont pas reproductibles.

1.2.1 Sources du jeu de test

Utilisation de sites web (S)

Avant la venue de NodeJS [2] en 2009, le langage JavaScript était principalement utilisé pour le développement de sites web (côté client). Il existait tout de même quelques usages plus niche du JavaScript dans d'autres contextes, comme WScript (JScript) [3], mais cela était négligeable. On retrouve beaucoup dans la littérature qui précède 2009 des jeux de test composés uniquement de sites web. Avant 2009, ce type d'éléments était représentatif de l'écosystème de code JavaScript, mais ce n'est pas tout à fait le cas aujourd'hui. On trouve maintenant du code JavaScript dans beaucoup d'autres contextes (côté serveur avec NodeJS, application desktop avec Electron, MySQL supporte le JavaScript pour ses procédures stockées, etc.).

Dans la section d'analyse de beaucoup d'articles académiques qui utilisent des sites web comme jeu de test, seul le nom de domaine du site web analysé est

1. `https://en.wikipedia.org/wiki/Alexa_Internet`
2. `https://nodejs.org/`
3. `https://en.wikipedia.org/wiki/Windows_Script_Host`

fourni dans l'article. On note souvent l'absence de date de visite, de copie digitale et même d'URL complète analysée. Ceci fait en sorte que ces articles ont des résultats non reproductibles, puisque :

— les sites analysés sont généralement dynamiques (aucune garantie d'obtenir le même code en revisitant le site ou en regardant un site d'archive comme archive.org)
— les sites analysés sont en développement continu (plusieurs années après la publication d'un article le site en question peut avoir complètement changé)

On voit, par exemple, ce constat dans l'article "Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications" (Ko *et al.*, 2015). Cet article présente dans ces résultats uniquement le nom de domaine des sites analysés sans préciser les pages visitées et leur date de visite. De plus, l'article ne cite aucun artéfact de recherche.

Utilisation de bibliothèques (L)

Dans la littérature, plusieurs articles scientifiques utilisent des bibliothèques ouvertes dans leur jeu de test. Même s'il ne s'agit pas nécessairement de programmes complets, ce type d'élément est intéressant pour plusieurs raisons :

— les bibliothèques ont souvent une quantité non négligeable de code
— les bibliothèques sont composées de code qui n'est pas synthétique
— lorsque les numéros de version sont indiqués, l'analyse est reproductible

Utilisation de lots de code venant de sources publiques (O)

Dans les articles scientifiques qui traitent de l'analyse de code JavaScript, quelques articles utilisent du code venant de sources publiques comme "Stack Overflow" [4] (Madsen *et al.*, 2015) ou encore le "Mozilla addon repository" [5] (Fass *et al.*, 2021). Un des principaux avantages de ce type de source est qu'il est facile de puiser un grand volume d'exemples. Par contre, ces exemples de code ne sont pas nécessairement représentatifs de code réel, puisqu'ils sont souvent rédigés par des développeurs non expérimentés. Il est aussi bon de noter que les articles qui utilisent ce type de source ont souvent des résultats non reproductibles, parce que les auteurs ne mentionnent pas toujours la liste exhaustive des éléments analysés et leurs versions. Même si la méthodologie de sélection des éléments est présente, le problème inhérent qui subsiste est que cette source de données change avec le temps.

Utilisation d'un banc de tests (B)

Quelques articles scientifiques utilisent pour l'analyse de leurs résultats des bancs de test. Un banc de test est un ensemble de tests qui visent à tester si une implémentation répond correctement à une ou plusieurs questions. Dans les articles trouvés, les bancs de tests ont la forme d'un ensemble de programmes ou de morceaux de code dont on connait certaines caractéristiques. Cette approche a le mérite d'offrir des résultats reproductibles. En principe, l'existence de ce type de jeu de test devrait permettre de pouvoir comparer les résultats de différents articles scientifiques. Par contre, très peu d'articles en utilisent. Sur l'ensemble des

4. `https://archive.org/details/stackexchange`

5. `https://addons.mozilla.org/en-US/firefox/extensions/`

articles scientifiques cités dans ce mémoire, seuls 9 en utilisaient au moins un et il ne s'agit pas toujours du même banc de tests. Une des raisons qui peut expliquer ce constat est la faible qualité des bancs de test ou le fait que certains ont un usage plutôt limité. Parmi les bancs de test utilisés, trois d'entre eux représentent la majorité de ce qui est utilisé.

IBM JavaScript Security Test Suite

Ce banc de test [6] se décrit comme contenant un grand nombre de tests unitaires qui présente une variété de défis que les analyseurs statiques doivent affronter. Par contre, cette affirmation est discutable considérant que le banc de test comporte 141 tests qui sont principalement des variantes de 23 catégories. Plusieurs catégories (ex. : Regex) font uniquement tester si certains API web sont supportés. De plus, tous les tests (sauf ceux des bibliothèques) font moins de 30 lignes de code JavaScript. Les seuls tests qui pourraient potentiellement évaluer la performance à grande échelle sont ceux des bibliothèques qui sont au nombre de trois. Par contre, ces trois tests n'indiquent aucune attente par rapport aux résultats à obtenir. Ainsi, si une approche termine rapidement, on ne peut pas distinguer s'il s'agit d'une approche performante en termes de temps d'exécution ou s'il s'agit d'une approche qui a fait une analyse trop partielle. Considérant que la performance sur des projets volumineux est un aspect important de l'analyse statique de vrais programmes, ceci est une lacune importante de ce banc de test.

Ce banc de test est utilisé dans Guarnieri *et al.* (2011).

---

6. `https://web.archive.org/web/20150924004127/https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1598`

Google Firing Range

Ce banc de test [7] est principalement destiné à évaluer le niveau de couverture des API web par les scanners de vulnérabilités. Les éléments de ce banc de test sont conçus de façon minimaliste (moins de 5 lignes de code JavaScript). Il est plutôt curieux de voir ce banc de test cité et utilisé dans un article académique, car il n'a pas du tout été conçu pour tester les limites de l'analyse statique de code JavaScript.

Ce banc de test est utilisé dans Parameshwaran *et al.* (2015).

ECMAScript Test Suite

Ce banc de test [8] est principalement destiné à évaluer si l'exécution de code JavaScript se fait selon le standard du langage. Dans le contexte d'analyse symbolique de code JavaScript, ceci est intéressant à utiliser pour s'assurer que ce que l'on développe interprète le code selon le standard. Il permet aussi d'offrir un comparatif de performance pour l'analyse symbolique. Par contre, en dehors de l'analyse de moteur d'analyse symbolique de code JavaScript, ce banc de test n'est pas très utile ou pertinent.

Ce banc de test est utilisé dans Santos *et al.* (2019) et Santos *et al.* (2018).

1.2.2 Sélection des éléments pour le jeu de test

Au-delà du type d'élément analysé, la sélection de ces éléments est pertinente à regarder.

---

7. `https://public-firing-range.appspot.com/`

8. `https://github.com/tc39/test262`

Les articles (Feldthaus *et al.*, 2013; Park *et al.*, 2016) ne mentionnent pas le processus de sélection des éléments qu'ils utilisent pour évaluer leur approche décrite dans leur article. Le principal risque de ce type d'approche est que les exemples choisis soient biaisés en faveur de l'approche décrite dans l'article.

Les articles (Parameshwaran *et al.*, 2015; Park et Ryu, 2015) ont une approche qui est moins susceptible à ce genre de biais en sélectionnant des éléments parmi des "top" maintenus par des sources indépendantes. On voit ceci beaucoup dans les articles qui utilisent des sites web dans leur section d'analyse. Ces articles utilisent principalement Alexa [9] comme référence. On voit aussi des articles qui citent la popularité des bibliothèques choisies comme critère d'inclusion.

1.2.3 Tailles des échantillons

Quelques articles analysent leurs résultats avec un nombre important d'éléments de test (>100). On observe par contre que la majorité d'articles trouvés font l'analyse de leur résultat sur moins de 100 éléments.

1.2.4 Variété des éléments

Quand on regarde les articles trouvés, on observe une grande variété de ce qui est testé pour évaluer la performance d'un article à l'autre. Au sein d'un même article, on n'observe pas cette même variété. Un des problèmes de ce point est qu'il est difficile d'extrapoler les résultats obtenus à l'ensemble des projets JavaScript. Une approche peut être performante dans un segment de projets JavaScript sans nécessairement l'être dans l'ensemble des projets.

Les segments de projets JavaScript qui ont été évalués dans les différents articles

---

9. `https://en.wikipedia.org/wiki/Alexa_Internet`

sont :

— Les pages web et leur code JavaScript
— Les bibliothèques JavaScript
— Du code JavaScript sans dépendances à des bibliothèques ou des cadriciels.
— Les extensions de navigateur web
— Du code JavaScript à source ouverte

Il est aussi bon de noter que, dans la variété de code testé, il y a très peu de gros projets. On note dans l'état de l'art que les cas choisis de test sont tout au plus dans une fourchette de 1 à 50,000 lignes de code. Aucune analyse avec des programmes plus gros n'a été présentée dans les articles de recherche qui ont été trouvés.

1.2.5 Variété des résultats

Comme l'analyse statique de code est fondamentalement un problème impossible à résoudre dans le cas général, on s'attendrait normalement à voir dans les résultats publiés d'un article une variété dans la précision obtenue et/ou la présence de cas de figure où l'analyse ne termine pas faute de temps ou d'espace mémoire. Ce qu'on observe malheureusement dans la littérature est que peu d'articles ont de la variété dans leurs résultats. Dans les articles cités dans ce mémoire, seuls les articles (Park et Ryu, 2015; Ko *et al.*, 2015; Park *et al.*, 2015; Julian *et al.*, 2012; Nielsen et Møller, 2020; Nielsen *et al.*, 2019; Li *et al.*, 2021; Kashyap *et al.*, 2014) présentent des cas où l'analyseur ne termine pas. De façon notable, certains articles comme (Feldthaus *et al.*, 2013; Anders *et al.*, 2009) ne présentent pas de cas où leur analyse ne termine pas. On voit aussi des articles qui excluent volontairement les résultats où l'analyseur a des enjeux (Park et Ryu, 2015).

1.2.6 Métriques utilisées

Une assez grande variété de métriques sont présentées dans les articles. Les plus communs sont sans surprise le taux de faux positif, le taux de vrai positif et le temps d'exécution. Malgré le fait que ces métriques sont plutôt de base, un bon nombre d'articles ne discutent pas de toutes ces métriques ou n'en discutent pas en terme précis (ex. : mentionne que l'analyse est faite avec un temps d'exécution maximal, mais ne donne pas le temps d'exécution des exemples ayant terminé).

Certaines métriques pertinentes, comme les faux négatifs, sont rarement discutées. Il faut dire que ce type de métrique est difficile à obtenir en pratique. La principale difficulté vient du fait qu'il faut avoir une source de vérité relativement complète pour établir l'ensemble des résultats qu'une analyse devrait produire et qui n'ont pas été produits.

On voit aussi des métriques comme la couverture de code être utilisées dans les articles qui traitent du sujet de l'analyse symbolique de code JavaScript.

1.2.7 Qu'est-ce qui détermine l'exactitude des résultats ?

Dans les articles de recherche, on voit trois tendances sur la façon dont on évalue si un résultat donné par l'implémentation de leur approche est exact ou non.

Dans la première tendance, on a les articles qui évaluent si leur approche permet de trouver les vulnérabilités présentes dans un corpus de code. On considère qu'un résultat est un vrai positif, si celui-ci fait partie des vulnérabilités attendues et on considère un résultat faux positif s'il ne fait pas partie des vulnérabilités attendues. Cette approche a le défaut de parfois évaluer davantage la qualité des règles utilisées avec l'approche plutôt que la capacité de l'approche lui-même. Une approche peut, par exemple, identifier qu'il existe un chemin par lequel une valeur

non sécurisée peut atteindre un appel clé. Cette observation peut être exacte sans qu'une vulnérabilité existe, puisqu'il peut y avoir des conditions autres que l'atteignabilité qui font en sorte qu'une vulnérabilité existe. Dans ce cas-ci, l'approche a correctement identifié qu'une des conditions nécessaires à la présence d'une vulnérabilité (l'atteignabilité) était présente. L'approche a donc partiellement raison d'inclure ce résultat. Cette approche d'évaluation a donc le défaut de considérer comme faux positif des résultats partiellement valides.

Dans la deuxième tendance, on a les approches qui évaluent les résultats en les comparant avec des traces d'exécution dynamiques. Dans ce type d'approche, c'est principalement l'outil Jalangi (Sen *et al.*, 2013) qui est utilisé pour obtenir des traces d'exécution. Ce type d'approche permet de bien évaluer le pourcentage de vrais arcs qui sont trouvés ou sont manquants. Ceci est principalement dû au fait que la trace d'exécution dynamique est complète (sans faux positif) comme source d'information. Si c'est présent dans la trace d'exécution dynamique, alors c'est nécessairement un arc possible. Là où cette approche est plus faible est pour l'analyse des faux positifs. En analyse dynamique, il est pratiquement impossible de déterminer que l'on a atteint tout le code qu'il est possible d'atteindre. Donc, on ne peut pas avoir de certitude que la base de référence obtenue par analyse dynamique contient tous les arcs possibles. Ceci fait en sorte qu'un résultat qu'un outil trouve et qui n'est pas présent dans la base de référence peut être un faux positif ou un vrai positif. Les articles de recherche doivent se rabattre sur une approche de la troisième catégorie pour évaluer les faux positifs avec tous les inconvénients qui viennent avec. Ce type d'analyse est complète, étant donné qu'il s'appuie sur des chemins qui sont garantis d'être possible en pratique, mais n'est pas conservatrice puisqu'elle exclut tous les chemins que l'exécution dynamique n'a pas atteints.

Dans la troisième tendance, on a les approches qui évaluent les résultats en éta-

blissant leur vérité de référence selon des critères d'évaluation. Ces critères servent avant tout à établir si un résultat d'un outil doit être considéré comme étant vrai ou non. Cette évaluation ne compare pas les résultats d'un outil avec une liste finie d'éléments qu'il est attendu de trouver. Elle évalue seulement si le résultat d'un outil respecte les critères établis. Le principal enjeu de ce type d'approche est que les critères choisis n'établissent pas nécessairement une vérité de référence qui est toujours vraie. Si, par exemple, on considère qu'un résultat est vrai si un chemin dans le code est possible sans égard aux structures de contrôle, on peut considérer des résultats comme étant vrais, alors qu'ils sont impossibles en pratique. Ces approches vont, pour cette raison, avoir tendance à établir une vérité de référence partiellement fausse ou incomplète.

1.3 Approche existante d'analyse statique de code JavaScript

1.3.1 Approches principales

À très haut niveau, l'approche qui est étudiée dans ce mémoire est divisible en deux étapes. La construction d'un graphe qui représente le code d'un projet et l'analyse de ce graphe pour identifier s'il existe des flux de données entre des puits et des sources. Certains articles de recherche portent uniquement sur la construction de graphe qui représente le code, alors que, d'autres proposent et/ou utilisent des approches qui couvrent les deux étapes.

Pour ce mémoire, nous nous sommes limités à l'analyse des articles qui traitent spécifiquement d'analyse de JavaScript. Il est tout de même bon de noter que des approches publiés pour l'analyse de d'autres langages peuvent aussi être applicables pour l'analyse de code JavaScript. Un bon exemple de ceci est l'outil

"Google Closure Compiler" [10] qui implémente en partie l'approche décrite dans l'article (Nilsson-Nyman *et al.*, 2009). Cet article traite d'analyse de code Java, mais est tout de même pertinent pour l'analyse de code JavaScript, puisqu'il a pu être utilisé dans cet outil d'analyse de code JavaScript. L'exclusion de ces articles fait en sorte que l'analyse faite ne couvrira pas nécessairement l'ensemble des approches applicables.

Approches de construction de graphe

Ce qui est intéressant dans l'état de l'art est qu'il y a une variété de représentations de code sous forme de graphe. Certaines représentations sont créées et utilisées pour des besoins spécifiques (ex. : l'identification de vulnérabilité de pollution de prototype (Arteau, 2018)), alors que, d'autres sont plus généralistes. On peut classer ces approches dans les catégories suivantes.

Construction de graphe d'appel

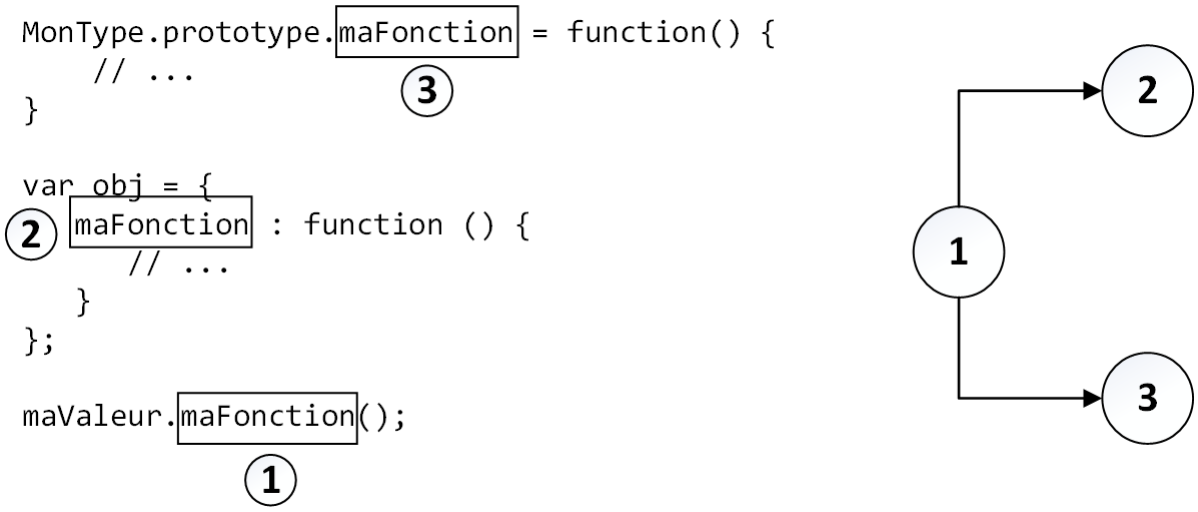
Les graphes d'appel modélisent la relation appelant/appelé entre les fonctions d'un projet. Ils représentent les fonctions qu'une fonction peut atteindre et par quelles fonctions une fonction peut être atteinte. Les approches pour construire ces graphes sont assez variées.

Dans les approches plus simplistes, on a celle décrite dans Efficient Construction of Approximate Call Graphs for JavaScript IDE Services (ACG) (Feldthaus *et al.*, 2013) qui propose de construire le graphe d'appel en se basant uniquement sur le nom des propriétés et des fonctions sans égard à la sémantique d'utilisation (sauf pour les cas triviaux). Ainsi, un appel à "monObjet.maFonction()" générerait un arc entre cet appel de fonction et toutes les fonctions nommées "maFonction"

10. `https://developers.google.com/closure/compiler`

dans le même projet. L'approche n'essaie pas d'établir ce lien en observant les valeurs possibles de "monObjet", il l'établit purement sur l'observation qu'une fonction de même nom est une approximation intéressante sans faire davantage de validation. La figure 1.1 montre l'approche de ACG.

FIGURE 1.1 – Fonctionnement de ACG



Dans les approches plus complexes, un concept récurrent est la notion de flux de données. Cette notion est importante pour l'analyse de JavaScript, parce qu'en dehors des cas triviaux, il est difficile de limiter les cibles possibles d'un appel de fonction, puisqu'il n'y a pas de types associés aux variables et il est aussi possible de créer des types dynamiquement à l'exécution. Ainsi, lorsqu'on observe le code "monObjet.maFonction()", il n'est pas possible de faire une inférence plus précise que $\top$ comme cible de cet appel sans faire d'analyse sur ce que "monObjet" peut contenir ou avoir une approche approximative comme ACG.

La notion de flux de données est, dans ce contexte-ci, beaucoup utilisée pour amener des bornes sur ce que "monObjet" peut contenir. Sans ces bornes, il est difficile de faire une analyse qui ne retourne pas tout le temps $\top$ ou $\bot$. Pour la construc-

tion d'un graphe d'appel, $\top$ et $\bot$ sont deux états particulièrement problématiques pour des raisons différentes. Lorsque l'on retourne $\top$, on crée incorrectement beaucoup trop de liens et notre taux de faux positifs est élevé. Lorsqu'on retourne $\bot$, on n'identifie aucune fonction cible possible. Dans la littérature, on voit donc des articles où les approches décrites obtiennent par propagation des informations qui permettent de contraindre ce que "monObjet" peut être dans ce contexte.

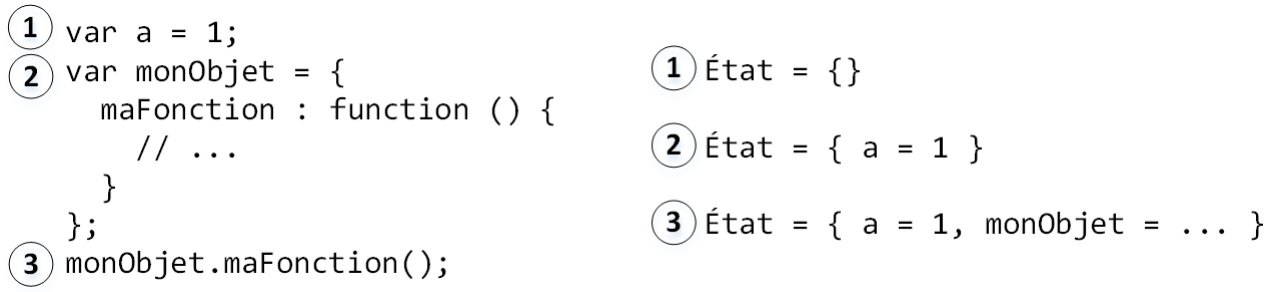
Par exemple, dans "Type Analysis for JavaScript" (TAJS) (Anders *et al.*, 2009), on commence avec un graphe sommaire basé sur l'AST du code et on fait une analyse par propagation pour identifier les arcs des appels interprocéduraux.

On remarque dans les articles trouvés que l'analyse de flux de données est parfois très près de l'analyse symbolique. Ces analyses de flux de données conservent un niveau variable d'information sur l'état des valeurs possibles à chaque instruction par laquelle il propage de l'information. On a, par exemple, dans l'article "Mining Node.js Vulnerabilities via Object Dependence Graph and Query" (Li *et al.*, 2022) une analyse de flux de données où on propage un niveau très important d'informations et qui émule le fonctionnement des fonctions natives de JavaScript. SAFE [11] propose le même type d'approche sans simuler de façon aussi précise l'exécution du code. Tout dépendant des paramètres utilisés, SAFE va émuler les boucles ou non, conserver l'ensemble des valeurs possibles ou non, etc. La figure 1.2 montre comment l'analyse de flux de données analyse ligne par ligne le code en propageant l'information sous forme d'états.

---

11. `https://github.com/kaist-plrg/safe`

FIGURE 1.2 – Fonctionnement de l'analyse par propagation



Dans l'analyse de flux de données, on voit aussi une notion de contrainte sur les valeurs contenues dans l'état. Ce type de contrainte est principalement présent pour éviter l'explosion combinatoire des valeurs dans certains contextes. SAFE, par exemple, permet de configurer à travers le paramètre "maxStrSetSize" un maximum de valeurs possibles qu'une variable de type String peut contenir. À partir du moment où ce maximum est atteint, SAFE considère la valeur comme étant un "String" avec $\top$ comme valeur possible.

On note aussi dans l'état de l'art des approches qui servent à représenter une portion spécifique du graphe d'appel. Par exemple, dans l'article "An Asynchronous Call Graph for JavaScript" (Seifert *et al.*, 2022), on s'attarde à la représentation des graphes d'appels pour uniquement les appels de fonctions qui utilisent la sémantique asynchrone de JavaScript.

Construction de graphe de propriétés de code

Le concept de CPG (graphe de propriétés de code – code property graph) a été introduit dans l'article "Modeling and Discovering Vulnerabilities with Code Property Graphs" (Yamaguchi *et al.*, 2014). Le concept derrière le CPG est d'unir les informations de l'AST du code, le graphe de flux de contrôle et le graphe de dépendance de programme dans un seul graphe. Il est bon de noter que, même

si l'article présente l'approche du CPG pour JavaScript, l'article ne décrit pas d'approche qui permet de construire un graphe d'appel pour JavaScript à partir du CPG. L'identification des vulnérabilités qui a été faite pour démontrer l'utilité du CPG a été faite sur du code C et non JavaScript. Pour pouvoir construire un graphe d'appel qui gère les cas non triviaux, il faudrait probablement appliquer une approche comme celle décrite dans TAJS ou SAFE sur ce graphe.

Dans la littérature, ce terme et concept est uniquement utilisé lors de discussion qui porte sur l'article de Yamaguchi *et al.* (2014) ou l'outil Joern qui implémente l'approche décrite dans l'article. Aucun autre article ne fait référence ou utilise ce concept.

Construction de graphe de propriétés d'objet

Le concept de graphe de propriétés a été amené par l'article "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis" (Li *et al.*, 2021). Ce type de graphe sert principalement à modéliser les liens entre les accès (en lecture et en écriture) aux propriétés d'un objet. Ce type de graphe est relativement niche, car il a été principalement mis de l'avant pour faire de la détection de vulnérabilité de pollution de prototype en JavaScript.

1.3.2 Utilisation du concept de partitionnement des traces

Le partitionnement des traces a été introduit par l'article "The Trace Partitioning Abstract Domain" (Rival et Mauborgne, 2007). La technique est mathématiquement définie comme étant une disjonction d'état choisie en fonction de critères de contrôle. L'idée générale derrière le partitionnement de trace, c'est de ne plus considérer tous les états possibles ensemble, mais de les considérer de façon disjointes. Les critères de contrôle déterminent quand on doit faire une disjonction (considérer un état de façon disjointe).

Si on considère le code de la figure 1.3, on ne considère plus simplement que a peut avoir la valeur 5 ou -5, on considère qu'il y a deux analyses possibles ($a = 5 \wedge b = 1$ et $a = -5 \wedge b \neq 1$). Ce partitionnement permet une analyse plus précise, parce que l'on ne considère plus l'état où $b \neq 1 \wedge a = 5$ et $b = 1 \wedge a = -5$.

FIGURE 1.3 – Exemple de code pour le partitionnement de trace

```
if (b == 1) {
    a = 5;
} else {
    a = -5;
}
```

Dans les articles plus récents qui discutent d'analyse statique de code, on voit ce concept être raffiné et appliqué à l'analyse statique de code JavaScript. L'article "Value Partitioning : A lightweight approach to relational static analysis for JavaScript" (Nielsen et Møller, 2020) discute de comment le partitionnement de trace peut être utilisé pour améliorer l'analyse des accès de propriété sur un objet. L'article propose l'idée que chaque trace a des noms de propriétés distincts et que l'analyse est plus précise en prenant en compte chaque nom de propriété individuellement plutôt que de les considérer comme un ensemble de valeurs possible.

Des approches d'analyse statiques comme celle décrite dans "JSAI : A Static Analysis Platform for JavaScript" (Kashyap *et al.*, 2014) utilisent le concept de partitionnement pour éliminer l'exploration d'états qui ont déjà été analysés.

1.3.3 Approches connexes

Plusieurs articles de recherche portent sur des sujets connexes à ce qui a été présenté dans la section des approches principales. Même s'ils ne présentent pas

d'approches qui font de la construction de graphe ou de l'analyse de flux de données, les éléments qu'ils présentent peuvent améliorer des approches existants de construction de graphe.

Approches de gestion des boucles

L'article "Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity" (Park et Ryu, 2015) introduit le concept de sensibilité aux boucles dans l'analyse de SAFE. Le concept de sensibilité aux boucles dans ce contexte est de considérer chaque itération indépendamment au lieu de considérer l'ensemble des valeurs possibles de toutes les itérations en même temps.

Approches de simplification ou transformation de code

On voit aussi dans la littérature des articles qui discutent de la réécriture de code pour divers objectifs. L'idée de ces transformations n'est pas nécessairement de faciliter l'analyse statique de code, mais les transformations qu'ils proposent peuvent avoir un impact sur l'analyse statique de code. On voit, par exemple, dans l'article "Remedying the Eval that Men Do" (Jensen *et al.*, 2012) une méthodologie pour réécrire le code qui utilise la construction "eval". L'objectif de l'article n'est pas la simplification de l'analyse de code, mais pourrait être utilisé dans cette optique.

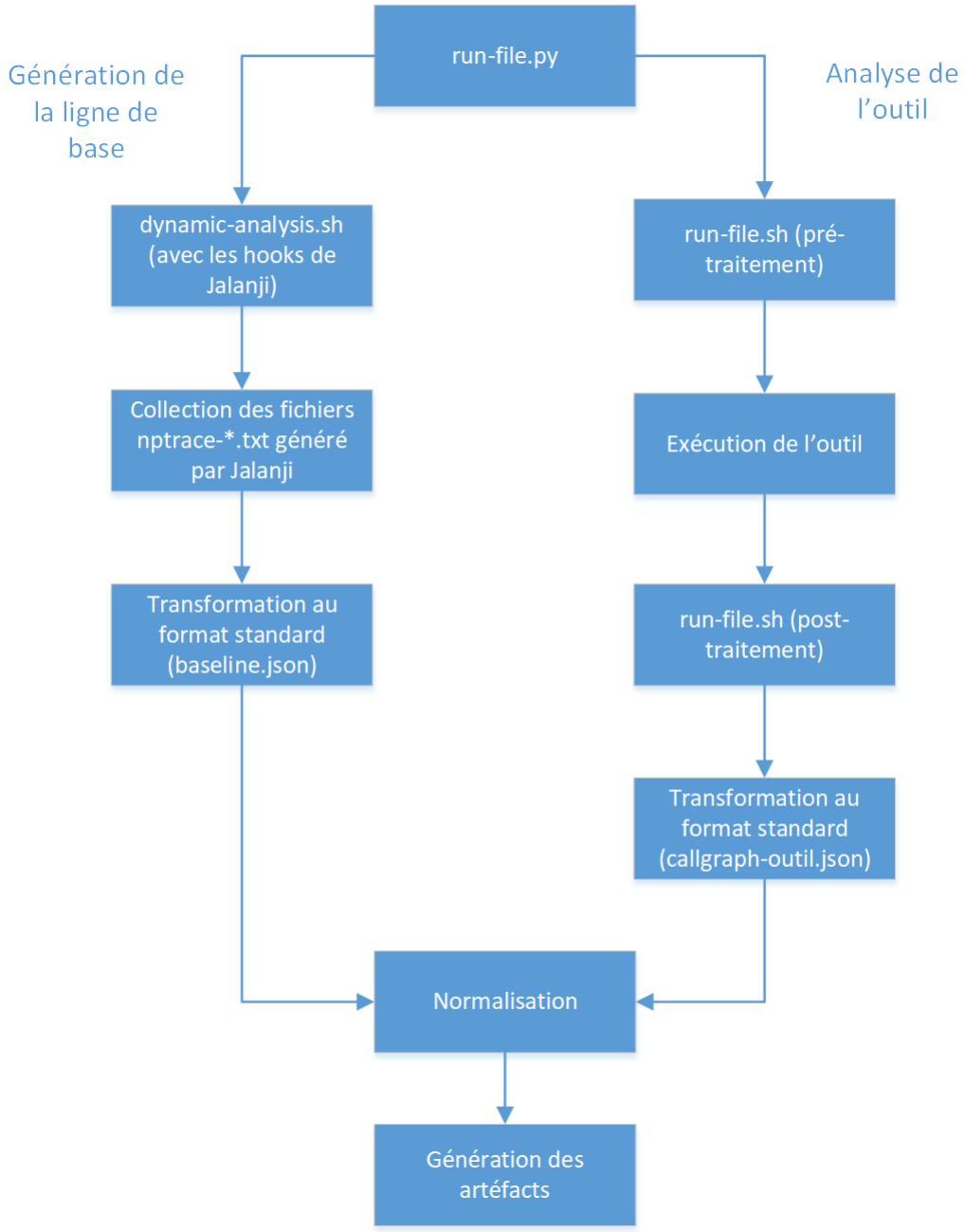
CHAPITRE II

MÉTHODOLOGIE

2.1 Vue d'ensemble

La figure 2.1 montre à haut niveau l'approche méthodologique utilisée pour obtenir le résultat des expérimentations. D'un côté, on a une analyse dynamique qui extrait la ligne de base pour établir une vérité sur les arcs possibles et de l'autre, on a l'exécution de l'outil et la collecte des résultats de l'outil. Les résultats des deux côtés sont ensuite normalisés pour être comparables et être capables de produire les artefacts de comparaison.

FIGURE 2.1 – Vue d'ensemble de l'approche méthodologique



2.2 Analyseurs sélectionnés

Ce projet de recherche s'est concentré sur les principaux outils académiques ayant une implémentation fonctionnelle. Cette sélection a mené au choix de WALA (v1.6.3), SAFE (v2.0), ACG (v1.3.2) et TAJS (v0.20). Comme ces quatre outils couvrent une variété intéressante d'approches d'analyse statique de code pour le JavaScript, il a été jugé suffisant de se limiter à ceux-ci. L'idée du projet de recherche n'est pas de faire une analyse exhaustive de tous les outils, mais bien de développer une méthodologie qui permet d'actualiser leurs résultats avec des projets modernes.

2.2.1 SAFE

SAFE est un outil qui a été principalement conçu pour produire le graphe de flux de contrôle d'un programme JavaScript. L'outil a été légèrement modifié [1] pour afficher un graphe d'appel à partir de l'algorithme qui produit le graphe de flux de contrôle. La version utilisée pour ce mémoire comprend l'implémentation de diverses approches décrites dans plusieurs articles différents pour améliorer la production d'un graphe de flux de contrôle.

Cet outil n'est plus maintenu depuis 2021.

2.2.2 ACG

ACG est un outil qui implémente l'approche de construction de graphe d'appel décrite dans l'article "Efficient construction of approximate call graphs for JavaScript IDE services" (Feldthaus *et al.*, 2013). Il existe plusieurs implémentations

1. `https://github.com/HoLyVieR/safe/commit/480e129b354`

de l'approche proposée dans cet article. L'implémentation qui est utilisé dans ce mémoire est celle nommé "js-callgraph" [2].

Cet outil n'est plus maintenu depuis 2019.

2.2.3 WALA

WALA est un outil d'analyse de code qui regroupe des algorithmes variés d'analyse de code pour JavaScript et Java. Le sous-ensemble des fonctionnalités de WALA qui est étudié dans ce mémoire est celui qui correspond aux algorithmes de génération de graphe d'appel pour le langage JavaScript.

Cet outil est encore activement maintenu.

2.2.4 TAJS

TAJS est un outil qui a été principalement conçu pour faire de l'inférence de type et produire le graphe d'appel d'un programme JavaScript.

Cet outil n'est plus maintenu depuis 2020.

2.3 Code à analyser sélectionné

Les deux principaux projets sélectionnés pour ce projet sont la bibliothèque underscore [3] (v1.13.6) et le projet GhostCMS [4] (v5.106.0).

---

2. `https://github.com/Persper/js-callgraph`

3. `https://github.com/jashkenas/underscore`

4. `https://github.com/TryGhost/Ghost`

La bibliothèque underscore a été choisie, parce qu'elle présente plusieurs caractéristiques intéressantes et qu'elle est de taille significative pour une bibliothèque (1 445 lignes de code JavaScript - statistique obtenues avec l'utilitaire cloc [5] sur le fichier "underscore-umd.js"). Les caractéristiques intéressantes de la bibliothèque sont le fait qu'elle n'a pas de dépendance à d'autres bibliothèques et que c'est fondamentalement du code "partiel" du fait que c'est une bibliothèque utilitaire. Le code de la bibliothèque underscore ne fait du travail utile que lorsqu'un projet l'intègre et fait des appels à ces fonctions.

Le projet GhostCMS a été choisi, parce qu'il présente des caractéristiques intéressantes et qu'il est de taille significative (256 576 lignes de code JavaScript - statistique obtenues avec l'utilitaire cloc [6]). Les caractéristiques intéressantes sont le fait qu'il utilise directement ou indirectement une énorme quantité de bibliothèques, qu'il s'agit d'un projet modulaire où les différents morceaux du projet sont dans des modules différents et qu'il s'agit d'un projet complet.

Il est bon de noter ici que le projet VS Code [7] a aussi été considéré puisqu'il présente aussi des caractéristiques similaires à GhostCMS. Par contre, comme l'intégration et l'analyse de gros projet est très exigante en temps et en effort, ce projet de recherche a choisi de se limiter à un seul gros projet.

2.4 Établir la ligne de base

Pour être capable d'établir si un résultat est un faux positif et identifier les faux négatifs, on doit établir une source de vérité. L'approche choisie pour ce projet

---

5. `https://github.com/AlDanial/cloc`

6. `https://github.com/AlDanial/cloc`

7. `https://github.com/microsoft/vscode`

de recherche est similaire à certains autres articles qui traitent d'analyse statique de code JavaScript où on utilise une analyse dynamique [8] pour extraire les appels de fonction qui s'effectuent lors de l'exécution du programme. Là où ce projet de recherche amène une approche légèrement différente est dans la façon dont cette idée a été adaptée pour être capable d'analyser de larges projets JavaScript. Une des principales difficultés de l'analyse dynamique dans cette approche est que la qualité des résultats est dépendante de la couverture de l'exécution dynamique. Dans un projet comme GhostCMS, simplement lancer l'application donnera une couverture relativement faible du code, parce que la majorité du code du projet est exécutée seulement lorsqu'il reçoit des requêtes HTTP.

Ce projet de recherche contourne cette difficulté en utilisant l'exécution des tests du projet. En surface, cette approche peut paraître simple, mais plusieurs obstacles sont présents.

Le premier est qu'il est difficile d'avoir un contrôle sur l'exécution de ces tests. Ceci est principalement dû au fait que les tests unitaires dans les projets sont généralement faits pour être exécutés à partir d'utilitaires comme "npm" et "yarn". Ainsi pour lancer les tests unitaires, notre commande initiale est "npm test" ou "yarn test". Ces commandes ne permettent pas de spécifier de paramètres additionnels à "node" ou de changer l'exécutable "node" utilisé. De plus, même si les tests sont initiés avec "npm test" ou "yarn test", ces commandes encapsulent une multitude de cadriciels qui ont un fonctionnement particulier et des configurations particulières. La seule constante est que l'exécutable "node" sera éventuellement lancé. Ce qui est problématique ici est que, pour extraire des traces dynamiques avec un outil comme Jalangi, on doit être capable de lancer le processus "node" avec des arguments spécifiques. Or, ici c'est "npm", "yarn" ou le cadriciel de test qui

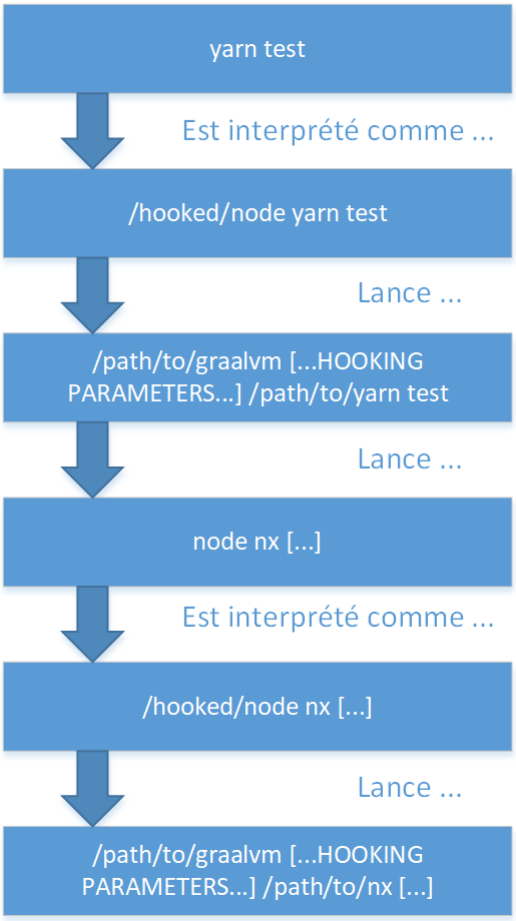
8. `https://github.com/Haiyang-Sun/nodeprof.js/`

lancent les processus que l'on veut observer. Pour résoudre ce problème, ce projet de recherche utilise une approche par remplacement de la variable d'environnement PATH. Dans cette approche, la variable d'environnement PATH pointe en priorité vers un dossier qui contient un script "node" dont l'invocation va démarrer le script passé en argument avec Jalangi. Cette approche fait en sorte que l'on est capable d'analyser avec Jalangi tous les scripts JavaScript qui seront invoqués par une commande. Ainsi, il devient possible d'obtenir l'ensemble des traces de toutes les invocations à "node" pour une commande comme "npm test".

Le deuxième est causé en partie par la solution du premier problème. Si on extrait les traces de toutes les exécutions, il va nécessairement avoir des traces d'exécution qui ne sont pas en lien avec le projet que l'on essaie réellement d'analyser. Ceci est toujours vrai pour "npm" et "yarn", car ces utilitaires sont écrits en JavaScript. Si on s'accroche à toutes les exécutions de code JavaScript, on va aussi s'accrocher à l'exécution de l'utilitaire "npm" et "yarn". Il est aussi bon de noter que comme on utilise l'exécution des tests pour établir notre ligne de base, cette exécution va aussi contenir l'exécution du cadriciel de test. Toutes ces données ne devraient pas faire partie de notre ligne de base puisqu'elles n'ont rien à voir avec l'exécution du projet. Pour résoudre ce problème, ce projet ajoute une étape après l'extraction des traces pour filtrer les données. Cette étape de filtrage enlève les nœuds et les arcs qui ne concernent pas des éléments présents dans les fichiers cibles.

L'approche proposée pour établir la ligne de base peut être résumée avec le schéma de la figure 2.2 qui montre comment l'analyse dynamique arrive à s'effectuer même si l'utilitaire qui roule les tests n'est pas "yarn", mais "nx".

FIGURE 2.2 – Cheminement de l'exécution lorsque la commande "yarn test" est analysée par l'analyse dynamique



2.5 Normalisation des résultats

Ce qui rend difficile la comparaison des résultats d'outils est que chaque outil produit des résultats différents à plusieurs égards. Le format (ex. : JSON, texte, etc.) et le schéma des données varient, mais il y a aussi d'autres aspects qui demandent plus de travail pour être capable de faire une analyse comparative. Parmi ces aspects, on retrouve le fait que les outils n'ont pas les mêmes références pour les nœuds des graphes d'appel. On fait parfois référence à une fonction en

fonction de la position du début de la déclaration `function (){...}` et d'autre fois on fait référence à une fonction en fonction de la position du début du corps de la fonction. Par exemple, pour le code `function foobar(){ }`, certains outils référencent la fonction "foobar" comme étant une fonction à la position 0 (début de la déclaration) et d'autres outils référencent la fonction "foobar" comme étant une fonction à la position 19 (début du corps). Ces références sont fondamentalement les mêmes, mais ils ont des libellés de nœuds différents puisque la position dans le libellé est différente. Pour cela, une étape de normalisation des résultats est nécessaire pour que la sortie de chaque outil soit comparable et analysable. Pour ce projet de recherche, les étapes de normalisation suivantes ont été effectuées :

— Les références des fonctions (les nœuds du graphe d'appel) ont toutes été transformées pour respecter un format qui concatène le chemin absolu du fichier JavaScript et la position absolue du début de la fonction dans le fichier.
— Chaque nœud du graphe d'appel se fait assigner un identifiant numérique qui est le même pour l'analyse dynamique et pour le résultat de l'outil. Ainsi un arc "1" vers "2" dans un outil est nécessairement équivalent à un arc "1" vers "2" présent dans l'analyse dynamique.
— Les nœuds et les arcs qui ne concernent pas le ou les fichiers analysés sont retirés. Ces nœuds et arcs sont considérés comme du bruit puisqu'ils ne sont pas associés au projet (ex. : code du cadriciel de test).

2.6 Analyse d'outils qui supportent uniquement l'analyse de fichier unique

Certains outils analysés dans ce projet de recherche (ex. : SAFE) supportent uniquement l'analyse de contenu JavaScript présent dans un fichier unique. Pour utiliser ces outils avec des projets comme GhostCMS, on doit avoir une approche

où on est capable de transformer un dossier de projet en un seul fichier JavaScript. Il existe des outils comme webpack qui sont capables d'effectuer cette tâche. Par contre, un des gros enjeux de l'utilisation de webpack c'est que le code combiné est significativement plus difficile à analyser statiquement. Ceci est principalement dû au fait que la ré-écriture des "require" se fait en associant des nombres à des fichiers ou paquetages et que les "require" sont remplacés par des références numériques. Comme l'objectif de ce projet de recherche n'est pas d'évaluer la capacité des outils d'analyse statique de code à comprendre la logique de webpack, l'approche d'utiliser webpack n'a pas été retenue. D'autres outils du même genre existent, mais exhibent ce même type de problème. C'est pour cette raison qu'une autre approche a été retenue. Cette approche consiste à faire une écriture combinée plus simpliste. Fonctionnellement, chaque fichier JavaScript est mis dans une fonction qui a un préambule pour déclarer que "export" est un objet et un épilogue qui retourne le contenu de "export". Les appels à "require" sont substitués à un appel de la fonction qui contient le contenu du fichier indiqué par "require". Il est bon de noter que cette approche n'est pas strictement sémantiquement équivalente au vrai comportement de "require", parce que "require" met en cache l'état d'un fichier une fois chargé. Ainsi on a normalement l'égalité suivante qui est vraie avec NodeJS `require('./module')=== require('./module')`. Cette égalité n'est plus présente avec cette approche pour combiner les fichiers JavaScript. Pour que cette égalité soit vraie, le même objet doit être retourné. Des instances différentes avec les mêmes clés et valeurs ne satisfont pas cette égalité. Aussi, cette approche ne supporte pas l'utilisation de "require" avec du contenu dynamique. Malgré ces défauts, cette représentation reste suffisamment proche de la réalité pour être utile sans introduire une complexité d'analyse importante. La complexité d'analyse reste faible ici, car les "require" sont substitués par des appels directs à des fonctions et il n'y a pas d'indirection présente.

La figure 2.5 présente comment le fichier de la figure 2.3 et le fichier de la figure 2.4 sont mis ensemble.

FIGURE 2.3 – Fichier index.js

```
var abc = require('./abc');
abc.faireQuelqueChose();
```

FIGURE 2.4 – Fichier abc.js

```
exports.faireQuelqueChose = function () {

};
```

FIGURE 2.5 – Fichier combiné

```
function projet_index() {
    var exports = {};
    var abc = projet_abc();
    abc.faireQuelqueChose();
    return exports;
}

function projet_abc() {
    var exports = {};
    exports.faireQuelqueChose = function () {

    };
    return exports;
}

projet_index();
```

2.7 Analyse d'outils qui ne supportent pas les versions récentes de JavaScript

Les outils présentés dans les différents articles de recherche ne sont pas tous activement maintenus. Ce qui est particulièrement difficile avec cet aspect est que la spécification de JavaScript (EcmaScript) a eu plusieurs révisions dans les dernières années qui ont ajouté, en autre, de la nouvelle syntaxe. Si on essaie de faire analyser le code de projet moderne avec ces outils, on a souvent comme problème que l'outil n'est pas capable d'obtenir l'AST du code. La nouvelle syntaxe ajoutée au langage n'introduit pas nécessairement quelque chose de plus complexe à analyser, mais elle fait en sorte que toutes les bibliothèques et outils qui extraient l'AST du code ne sont plus fonctionnels s'ils ne sont pas mis à jour. Pour résoudre ce problème, une approche retenue a été utilisée est d'utiliser le transpilateur Babel [9]. Ce transpilateur peut transposer du code JavaScript qui utilise des fonctionnalités récentes en du code JavaScript équivalent qui n'utilise pas de syntaxe récente du langage. Comme la transposition garde la sémantique du code original, on peut analyser le code transposé et cette analyse est équivalente à l'analyse du code sans transposition. Cette approche nous permet donc d'analyser du code récent avec de plus vieux outils. Le seul enjeu notable avec l'utilisation de Babel est dans la façon qu'il transpose l'usage de certains API ou syntaxes récentes. Dans certains cas, Babel va rajouter des fonctions utilitaires au début du code qui vont être appelées dans le code pour émuler l'API ou la syntaxe récente. Ceci peut en partie changer le graphe d'appel réel du projet. Ainsi, un outil peut correctement identifier des arcs entre le code du projet et les fonctions utilitaires de Babel. Or, ces arcs n'existent pas dans le projet original. Il est difficile d'automatiser la correction des déformations que cela amène à l'analyse.

Une autre approche évaluée a été de mettre à jour l'outil évalué pour supporter

9. `https://babeljs.io/`

la syntaxe récente. Cette approche peut être relativement complexe à appliquer, étant donné qu'il faut non seulement mettre à jour l'extraction de l'AST, mais aussi que les nouveaux éléments de syntaxe soient supportés dans l'analyse. Cette approche a uniquement été évaluée avec js-callgraph (ACG).

2.8 Gestion des transformations de code

Pour supporter les outils qui supportent l'analyse de fichier unique et les outils qui ne supportent pas les versions récentes de JavaScript, ce projet de recherche fait des transformations sur le code original avant de l'envoyer à l'outil. Ceci amène un problème important pour l'analyse des résultats. Les résultats de l'analyse dynamique vont pointer sur des lignes et/ou positions dans les fichiers originaux, alors que les résultats des outils vont pointer sur des lignes et/ou positions dans les fichiers transformés. Pour un résultat qui est le même des deux côtés, les valeurs identifiées peuvent être différentes si les transformations sont substantielles. On ne peut ainsi pas faire une correspondance directe entre les deux avec ces approches. Pour résoudre ce problème, ce projet de recherche utilise le concept de "correspondance de source" (source mapping) pour transformer les résultats des outils. Cette transformation va essentiellement regarder tous les nœuds des résultats de l'outil et utiliser les fichiers de correspondance pour établir à quelle position le nœud correspond dans le code original. Dans le cas de la transposition avec Babel, les fichiers de correspondance utilisés sont les fichiers de source map [10] standard que Babel peut générer avec l'option "-s true". Pour la transformation qui permet de grouper les fichiers d'un dossier dans un seul fichier, il a fallu produire un fichier qui permet de retrouver les positions originales. Une fois cette correspondance faite entre les fichiers transformés et originaux, les résultats peuvent maintenant

10. `https://tc39.es/ecma426/`

être comparés aux résultats de l'analyse dynamique.

Il est aussi bon de noter que l'analyse de contenu JavaScript avec des caractères Unicode multioctets a été source de plusieurs bogues durant le développement des scripts. L'interprétation des caractères Unicode multioctets peut différer d'un outil à l'autre et cela peut amener des erreurs où la position des éléments est décalée de 1 à 3 octets par rapport à sa vraie position dans le fichier. Si ce type d'erreur d'analyse se répète souvent dans le même fichier, il est possible d'accumuler des erreurs de positionnement assez importantes pour que les positions ne pointent plus sur les bonnes fonctions.

2.9 Modifications aux outils existants

Pour effectuer les analyses requises, certains outils ont dû être modifiés. Ces modifications étaient nécessaires pour soit supporter des éléments qui étaient préalablement non supportés ou pour permettre des analyses qui n'étaient pas possibles avec les paramètres existants de l'outil.

2.9.1 SAFE

Cet outil implémente l'approche décrite dans l'article (Lee *et al.*, 2012).

Au moment de faire ce travail de recherche, l'outil en question ne pouvait pas être compilé sans avoir d'erreur qui bloquait la compilation du projet. La source de ces erreurs était la version de Scala utilisée, ainsi que la version de SBT [11] utilisée. Un correctif a été fait pour avoir une combinaison fonctionnelle de Scala et SBT.

---

11. `https://www.scala-sbt.org/`

Pour rendre l'exécution plus facile à reproduire, des configurations Docker [12] ont été ajoutées au projet SAFE. Ces configurations permettent de compiler le projet dans un environnement Docker, ainsi que d'exécuter l'analyse dans un environnement Docker.

Pour prendre en compte que SAFE ne modélise pas entièrement l'environnement d'exécution JavaScript, une variation de l'outil a été faite dans la branche "master_and_patch_non_strict". Cette variation fait en sorte que l'accès à une variable non définie est considéré comme $\top$ au lieu de $\bot$. On veut ainsi considérer qu'un élément "non défini" l'est peut-être parce que SAFE n'est pas conscient de son existence.

Ces modifications ont été publiées sur GitHub [13].

2.9.2 js-callgraph (ACG)

Cet outil [14] implémente l'approche décrite dans l'article (Feldthaus *et al.*, 2013).

Au moment de faire ce travail de recherche, l'outil en question n'avait pas été mis à jour depuis 2019. Par le fait même, les nouvelles fonctionnalités de JavaScript n'étaient pas supportées. Plusieurs modifications ont été nécessaires pour pouvoir analyser des projets récents. L'engin utilisé pour obtenir l'AST a dû être changé de esprima [15] (dernière version date de 2018) vers acorn [16]. Cette migration était

---

12. https://en.wikipedia.org/wiki/Docker_(software)

13. https://github.com/HoLyVieR/safe

14. https://github.com/Persper/js-callgraph

15. https://github.com/jquery/esprima

16. https://github.com/acornjs/acorn

nécessaire, car esprima n'est plus maintenu et ne supporte pas les versions de JavaScript qui sont sorties depuis sa dernière mise à jour. De plus, le code de ce projet a dû être adapté pour supporter adéquatement les nouvelles syntaxes du langage. Par exemple, comme il est permis d'avoir des déclarations d'objet dont les clés sont des expressions, le code de ce projet a dû être adapté pour ne pas terminer en erreur et pour faire en sorte que les nœuds des clés soient correctement visités. Certains changements plus importants étaient aussi nécessaires pour ne pas manquer inutilement des nœuds et des arcs. Les révisions plus récentes de JavaScript ont amené le concept de propriétés privées (ex. : `this.#maPropPrivee`). Ces propriétés privées sont nommées différemment au niveau de l'AST que les propriétés régulières ("Identifier" v.s. "PrivateIdentifier"). Préalablement, le code analysait uniquement les "Identifier" pour trouver les nœuds et les appels. Il a donc été nécessaire d'ajouter le support des "PrivateIdentifier" à l'outil.

Ces modifications ont été publiées sur GitHub [17].

2.10 Exclusions des résultats

Les expérimentations faites dans ce mémoire ont été exécutées sur une machine ayant 32Go de RAM. Cependant, comme plusieurs analyses ont nécessité une quantité supérieure de RAM à 32Go, la machine a dû être reconfigurée avec 32Go de "swap" pour que davantage d'analyses puissent terminer. L'utilisation de "swap" pour une partie des expérimentations peut grandement fausser la qualité des données obtenues. C'est pour cette raison que l'analyse des résultats ne s'est pas faite par rapport à l'aspect performance des outils.

---

17. `https://github.com/HoLyVieR/js-callgraph`

2.11 Artéfacts

Une fois les outils exécutés, les informations suivantes sont collectées à partir du résultat produit par l'outil et du résultat de l'analyse dynamique qui établit la ligne de base.

2.11.1 Ligne de base

Fichier : baseline.json

La ligne de base correspond aux nœuds et aux arcs qui ont été identifiés par l'analyse dynamique. Chaque entrée dans ce fichier est soit un objet qui représente un nœud ou un tableau qui représente un arc entre deux nœuds identifiés par identifiant unique.

2.11.2 Ligne de base normalisée

Fichier : baseline-normalized.json

La ligne de base normalisée correspond au contenu de la ligne de base à la différence que les identifiants uniques des nœuds ont été réétiquetés de façon à ce qu'un nœud identique dans la ligne de base et dans le résultat de l'outil ait le même identifiant unique.

2.11.3 Résultats de l'outil

Fichiers : callgraph-*outil*-callgraph.json et callgraph-*outil*-callgraph-normalized.json

La sortie au format standard est stockée dans le premier fichier et le contenu normalisé est stocké dans le deuxième. Le processus de normalisation est celui

décrit dans la ligne de base normalisée.

2.11.4 Sortie standard et sortie d'erreur

Fichiers : callgraph-*outil*-raw.txt et callgraph-*outil*-raw-err.txt

Les messages de sortie des outils et scripts exécutés sont conservés dans les deux fichiers en question. Le premier conserve le contenu qui provient de la sortie standard et le deuxième conserve le contenu qui provient de la sortie d'erreur.

2.11.5 Comparatifs des nœuds et arcs

Fichiers : callgraph-*outil*-common-edges.json et callgraph-*outil*-common-nodes.json

Ces deux fichiers contiennent les arcs et les nœuds qui sont communs entre la ligne de base et le résultat de l'outil. Ils sont produits en faisant l'intersection de ligne de base normalisée et le résultat de l'outil normalisé.

Fichiers : callgraph-*outil*-extra-edges.json et callgraph-*outil*-extra-nodes.json

Ces deux fichiers contiennent les arcs et les nœuds qui sont présents dans le résultat de l'outil, mais qui ne sont pas présents dans la ligne de base. Ils sont produits en faisant la soustraction du résultat de l'outil normalisé avec la ligne de base normalisée.

Fichiers : callgraph-*outil*-missing-edges.json et callgraph-*outil*-missing-nodes.json

Ces deux fichiers contiennent les arcs et les nœuds qui sont présents dans la ligne de base, mais qui ne sont pas présents dans le résultat de l'outil. Ils sont produits en faisant la soustraction de la ligne de base normalisée avec le résultat de l'outil normalisé.

CHAPITRE III

EXPÉRIMENTATIONS

Dans cette section les expérimentations qui ont été exécutées sont décrites. Les résultats de ces expérimentations ont été publiés sur GitHub [1]. Chacune des expérimentations présentées a des caractéristiques et des objectifs de recherche qui sont expliqués. Le choix de quoi exécuter avec quel paramètre découle donc principalement des questions de recherche.

Pour alléger la présentation des résultats, les acronymes suivants ont été utilisés. "NC" indique le nombre de nœuds communs. "AC" indique le nombre d'arcs communs. "NS" indique le nombre de nœuds supplémentaires. "AS" indique le nombre d'arcs supplémentaires. "NM" indique le nombre de nœuds manquants. "AM" indique le nombre d'arcs manquants. "-" indique que la donnée n'a pu être obtenue.

Étant donné que la méthodologie utilisée permet d'établir une source de vérité qui est seulement un sous-ensemble des nœuds et des arcs possibles, il n'est pas possible de quantifier les valeurs obtenues en termes de vrai positif, faux négatifs, faux positifs et vrai négatif. Les "NC" et "AC" sont des vrais positifs, mais ne sont pas nécessairement l'ensemble des vrais positifs. Les "NS" et "AS" peuvent

1. `https://github.com/HoLyVieR/maitrise-2025-artefact-public`

être autant des vrais positifs que des faux positifs. Les "NM" et "AM" sont des faux négatifs, mais ne sont pas nécessairement l'ensemble des faux négatifs.

3.1 Expérimentations avec ACG

3.1.1 Expérience 1 (ACG - GhostCMS sans node_modules)

Pour cette expérience, l'outil ACG a été exécuté sur le projet de GhostCMS. L'outil a été configuré avec comme exclusion les dépendances externes (ce qu'on retrouve dans le dossier "node_modules") ainsi que les fichiers appartenant aux tests unitaires.

L'objectif général de cette expérience est d'observer la performance de l'outil ACG dans un contexte où seul le code du projet est directement analysé.

NC	AC	NS	AS	NM	AM
756 (19.1%)	493 (11.7%)	21 945	389 973	3 202 (80.9%)	3 704 (88.3%)

Tableau 3.1 – Sommaire des résultats de l'expérience 1

Le 19.1% de nœuds identifiés correctement et le 11.7% d'arcs identifiés correctement s'expliquent en grande partie par le fait qu'une proportion importante des nœuds et des arcs sont dans le code ou lié au code présent dans le dossier node_modules qui a été exclu. En omettant ce dossier, on coupe une bonne partie des résultats.

3.1.2 Expérience 2 (ACG - GhostCMS sans les dossiers test/)

Pour cette expérience, l'outil ACG a été exécuté sur le projet de GhostCMS. L'outil a été configuré avec comme exclusion les fichiers appartenant aux tests unitaires seulement.

L'objectif général de cette expérience est d'observer si l'approche de ACG permet de construire un graphe d'appel qui inclut aussi les dépendances du projet. Pour cette expérience, l'exécution a été manuellement arrêtée après 2 jours (l'expérience n'était toujours pas terminée). Il n'y a donc pas d'artefacts de sortie.

NC	AC	NS	AS	NM	AM
-	-	-	-	-	-

Tableau 3.2 – Sommaire des résultats de l'expérience 2

On peut conclure de cette expérience que l'exclusion de dossiers supplémentaires comme test/ ne suffit pas pour avoir une analyse qui termine.

3.1.3 Expérience 3 (ACG - GhostCMS avec fichiers de la ligne de base seulement)

Pour cette expérience, l'outil ACG a été exécuté sur une partie du projet de GhostCMS. La partie qui a été sélectionnée est composée uniquement des fichiers présents dans la ligne de base obtenue dynamiquement. Pour ce faire, tous les fichiers référencés dans la ligne de base ont été mis dans un dossier séparé et l'outil ACG a été exécuté sur ce dossier.

L'objectif général de cette expérience est d'observer la performance et la qualité des résultats de ACG dans un contexte où une analyse d'atteignabilité a été préalablement faite. La sélection des fichiers a donc été faite pour simuler cela. Comme l'objectif est d'évaluer avant tout ACG et non les différentes approches pour déterminer l'atteignabilité, cette approche synthétique a été jugée préférable.

NC	AC	NS	AS	NM	AM
2 918 (73.7%)	2 606 (62.1%)	12 515	136 590	1 040 (26.3%)	1 591 (37.9%)

Tableau 3.3 – Sommaire des résultats de l'expérience 3

On peut conclure de cette expérience que ACG performe relativement bien quand l'analyse est faite sans fichier superflu. L'analyse des nœuds et arcs supplémentaires est difficile à faire ici, car l'analyse dynamique utilisée avait une couverture de code très faible.

3.1.4 Expérience 4 - (ACG - underscore)

Pour cette expérience, l'outil ACG a été exécuté sur la bibliothèque underscore. Le fichier empaqueté ("underscore-umd.js") a été choisi principalement parce que les tests unitaires du projet utilisent spécifiquement ce fichier lors de l'exécution des tests. La trace dynamique de l'exécution des tests fait donc uniquement référence à ce fichier. Il est donc plus simple pour l'expérience et le comparatif des résultats que les outils analysent ce fichier au lieu de tout le dossier du projet.

L'objectif général de cette expérience est d'observer la performance et la qualité des résultats de ACG dans un contexte où ce qu'on analyse est une bibliothèque (code partiel).

NC	AC	NS	AS	NM	AM
138 (78.0%)	153 (45.4%)	23	58	39 (22.0%)	184 (54.6%)

Tableau 3.4 – Sommaire des résultats de l'expérience 4

On peut conclure de cette expérience que ACG performe relativement bien pour des projets sans dépendance à d'autres bibliothèques. L'analyse dynamique faite sur la bibliothèque underscore avait une excellente couverture de code, étant donné que les tests unitaires de la bibliothèque sont exhaustifs. Les nœuds et arcs supplémentaires notés ici sont ainsi fort probablement de réels faux positifs. On peut

voir que leurs nombres (23 nœuds et 58 arcs) sont relativement petits par rapport au nombre total de vrais éléments (177 nœuds et 337 arcs). L'approche d'ACG a généré peu de faux positifs dans ce cas-ci.

3.1.5 Expérience 5 - (ACG - pdf.js)

Pour cette expérience, l'outil ACG a été exécuté sur la bibliothèque "pdf.js" [2]. Ce choix a été fait principalement parce que cette bibliothèque a été analysée dans l'article original de ACG (Feldthaus *et al.*, 2013). Cette expérience cherche donc à vérifier que les résultats obtenus par l'approche mise de l'avant donnent des résultats similaires à ce qui avait été originalement obtenu. Ce comparatif peut être fait, car c'est la même version de pdf.js qui est utilisée dans cette expérimentation. Il est bon de noter que, même si certains artefacts de recherche ont été publiés, certains aspects importants comme les outils utilisés obtenir la ligne de base dynamique ainsi que des paramètres comme quel PDF a été analysé avec pdf.js étaient absents. On ne peut donc pas avoir une reproduction parfaite dans ce contexte. Ce qui a été choisi comme substitut de PDF est le fichier "helloworld.pdf" du projet pdf.js.

NC	AC	NS	AS	NM	AM
186 (99.5%)	226 (85.0%)	3 472	5 136	1 (0.5%)	40 (15.0%)

Tableau 3.5 – Sommaire des résultats de l'expérience 5

Les résultats obtenus ici sont compatibles avec les résultats de l'article de recherche de ACG (Feldthaus *et al.*, 2013).

---

2. `https://github.com/mozilla/pdf.js`

3.1.6 Expérience 6 (ACG - GhostCMS transformé)

Pour cette expérience, nous avons exécuté ACG avec la version transformée du projet GhostCMS qui a été utilisé dans l'expérience 7, 8 et 9. L'objectif de cette expérience est d'observer à quel point les transformations faites au projet affectent la capacité des outils à analyser le projet.

NC	AC	NS	AS	NM	AM
554 (14.0%)	70 (1.7%)	56 448	3 395 676	3 404 (86.0%)	4127 (98.3%)

Tableau 3.6 – Sommaire des résultats de l'expérience 6

Les résultats sont pires que dans l'expérience 1 (ACG - GhostCMS sans node_modules) et l'expérience 3 (ACG - GhostCMS avec fichiers de la ligne de base seulement) sur tous les aspects. Non seulement l'approche identifie beaucoup moins d'arcs (1.7% vs 11.7%/62.1%) et de nœuds (14.0% vs 19.1%/73.7%), mais beaucoup plus d'arcs et nœuds supplémentaires sont présents dans les résultats (3 395 676 arcs vs 389 973/136 590 arcs). Même s'il est difficile d'établir la proportion réelle de faux positifs dans le lot, avec une différence aussi marquée (8.7x et 24.9x), il est raisonnable de supposer qu'une majorité de ces résultats sont des faux positifs. Le seul point positif est que l'analyse a terminé, ce qui n'était pas le cas pour l'expérience 2.

3.2 Expérimentations avec SAFE

3.2.1 Expérience 7 (SAFE avec paramètre par défaut - GhostCMS)

Pour cette expérience, SAFE a été exécuté avec les paramètres par défaut sur le projet GhostCMS. L'objectif de cette expérience était d'observer le comportement de l'outil de base. Cette expérience a terminé avec une erreur indiquant un manque

de mémoire.

NC	AC	NS	AS	NM	AM
-	-	-	-	-	-

Tableau 3.7 – Sommaire des résultats de l'expérience 7

L'expérience montre que l'analyse du projet GhostCMS avec SAFE ne termine pas avec les paramètres par défaut.

3.2.2 Expérience 8 (SAFE maxStrSetSize - GhostCMS)

Pour cette expérience, SAFE a été exécuté avec les paramètres par défaut sauf pour le paramètre "maxStrSetSize=10" qui limite le nombre d'états possible conservé pour une valeur. Passé le seuil de 10 valeurs différentes possibles, une chaîne de caractère est considérée comme pouvant avoir n'importe quelle valeur ($\top$).

Cette variante a été choisie pour évaluer si le problème d'explosion combinatoire des valeurs observé lors de l'expérience 7 était un bloquant qui empêchait l'obtention de résultats. Lors de l'expérience 7, il avait été observé que le manque de mémoire découlait du fait qu'il y avait une explosion combinatoire d'états possibles pour les chaînes de caractères. SAFE finissait par essayer de gérer un nombre exponentiel d'états au point de planter à cause du manque de mémoire disponible. En limitant le nombre de valeurs possibles à 10, SAFE ne devrait plus rencontrer ce problème. La valeur 10 a été choisie par essai-erreur et s'est figée sur cette valeur, parce que c'est la plus grande valeur qui permet à l'expérience de terminer dans l'expérience 11.

Cette expérience a terminé avec une erreur indiquant un manque de mémoire. De plus l'analyse partielle des résultats semble montrer que SAFE s'est embourbé à

l'infini dans des appels fonctions.

NC	AC	NS	AS	NM	AM
-	-	-	-	-	-

TABLEAU 3.8 – Sommaire des résultats de l'expérience 8

L'expérience montre que l'analyse du projet GhostCMS avec SAFE ne termine pas avec les paramètres par défaut et "maxStrSetSize=10".

3.2.3 Expérience 9 (SAFE maxStrSetSize,loopIter,callsiteSensitivity - GhostCMS)

Pour cette expérience, SAFE a été exécuté avec des paramètres choisis pour spécifiquement éviter les analyses lourdes et couteuses. L'objectif de cette expérience est d'évaluer, si en enlevant tous les éléments lourds de l'analyse, on pourrait avoir des résultats et si oui, de quelle qualité. Les paramètres ont été choisis en fonction de ce qui a permis à l'expérience 10 de terminer.

- "maxStrSetSize=10" pour empêcher la croissant exponentiel des états dans les chaînes de caractères.
- "loopIter=1" pour désactiver la sensibilité aux boucles.
- "callsiteSensitivity=1" pour limiter la sensibilité au contexte.

Cette expérience a terminé avec une erreur indiquant un manque de mémoire. De plus l'analyse partielle des résultats semblent montrer que SAFE s'est embourbé à l'infini dans des appels fonctions.

NC	AC	NS	AS	NM	AM
-	-	-	-	-	-

TABLEAU 3.9 – Sommaire des résultats de l'expérience 9

L'expérience montre que l'analyse du projet GhostCMS avec SAFE ne termine pas avec les paramètres par défaut et "maxStrSetSize=10,loopIter=1,callsiteSensitivity=1".

3.2.4 Expérience 10 (SAFE avec paramètre par défaut - underscore)

Pour cette expérience, SAFE a été exécuté avec les paramètres par défaut sur la bibliothèque "underscore". Cette expérience a pour objectif d'établir la ligne de base du comportement de l'outil pour l'expérience avec cette bibliothèque. Cette expérience n'a pas terminé, car elle s'est mise à considérer une quantité exponentielle d'états dans l'analyse d'une des boucles.

NC	AC	NS	AS	NM	AM
-	-	-	-	-	-

Tableau 3.10 – Sommaire des résultats de l'expérience 10

L'expérience montre que l'analyse du projet underscore avec SAFE ne termine pas avec les paramètres par défaut.

3.2.5 Expérience 11 (SAFE maxStrSetSize,callsiteSensitivity - underscore)

Pour cette expérience, SAFE a été exécuté avec les paramètres "maxStrSetSize=10" et "callsiteSensitivity=1" sur le projet "underscore". Ici, ce que l'on veut observer, c'est surtout l'effet de "maxStrSetSize" sur la capacité de SAFE à terminer et de la qualité des résultats qui peuvent être obtenus. "callsiteSensitivity=1" a dû être ajouté, car l'analyse avec trop de contexte causait une explosion combinatoire d'états qui étaient explorés. Ce paramètre était donc nécessaire pour rendre l'expérience 11 et 12 possible.

NC	AC	NS	AS	NM	AM
17 (9.6%)	16 (4.7%)	1	5	160 (90.4%)	321 (95.3%)

TABLEAU 3.11 – Sommaire des résultats de l'expérience 11

L'expérience montre que l'analyse du projet underscore avec SAFE termine avec les paramètres par défaut et "maxStrSetSize=10,callsiteSensitivity=1". L'expérience montre que SAFE n'a pas été capable d'identifier un grand nombre de nœuds (9.6%) et arcs (4.7%). Par contre, on remarque la très faible quantité de nœuds et arcs supplémentaire, ce qui indique que SAFE a généré peu de faux positifs.

3.2.6 Expérience 12 (SAFE maxStrSetSize,callsiteSensitivity,loopIter - underscore)

Pour cette expérience, SAFE a été exécuté avec les paramètres "maxStrSetSize=10", "loopIter=1" et "callsiteSensitivity=1" sur le projet "underscore". Ici, ce que l'on veut observer, c'est surtout l'effet de "loopIter=1" sur la capacité de SAFE à terminer et de la qualité des résultats qui peuvent être obtenus. "callsiteSensitivity=1" a du être ajouté pour les mêmes raisons que l'expérience 11. "maxStrSetSize=10" a du être conservé, car d'autres parties de l'analyse de SAFE (autre que l'analyse de boucle) peuvent causer une explosion de valeur possible pour les chaînes.

NC	AC	NS	AS	NM	AM
17 (9.6%)	16 (4.7%)	1	5	160 (90.4%)	321 (95.3%)

TABLEAU 3.12 – Sommaire des résultats de l'expérience 12

L'expérience montre que le paramètre "loopIter=1" n'a eu aucun impact sur les

résultats de l'analyse de la bibliothèque underscore lorsque les paramètres "maxStrSetSize=10,callsiteSensitivity=1" sont présents. Les résultats sont identiques à l'expérience 11.

3.2.7 Expérience 13, 14 et 15 (SAFE - pdf.js)

Pour ces expérimentations, SAFE a été exécuté sur le projet pdf.js. L'objectif de ces expérimentations est de voir le comportement de SAFE pour des projets de plus petite taille avec des paramètres identiques à ce qui a été utilisé pour de plus gros projets dans l'expérience 10, 11 et 12.

NC	AC	NS	AS	NM	AM
15 (8.0%)	13 (4.9%)	0	0	172 (92.0%)	253 (95.1%)

Tableau 3.13 – Sommaire des résultats de l'expérience 13

NC	AC	NS	AS	NM	AM
15 (8.0%)	13 (4.9%)	0	0	172 (92.0%)	253 (95.1%)

Tableau 3.14 – Sommaire des résultats de l'expérience 14

NC	AC	NS	AS	NM	AM
15 (8.0%)	13 (4.9%)	0	0	172 (92.0%)	253 (95.1%)

Tableau 3.15 – Sommaire des résultats de l'expérience 15

Les résultats pour le projet pdf.js sont similaires à ceux obtenus avec underscore. SAFE identifie peu de nœuds et d'arcs, mais ceux identifiés sont tous valides.

3.3 Expérimentations avec WALA

3.3.1 Expérience 16 (WALA - underscore)

Pour cette expérience, l'outil WALA a été exécuté sur le projet underscore. L'objectif de cette expérience est d'observer les éléments que WALA a de la difficulté à analyser en pratique. Les expérimentations avec WALA ont aussi été faites pour avoir un peu plus de variété dans les outils choisis.

NC	AC	NS	AS	NM	AM
163 (92.1%)	267 (79.2%)	11	468	14 (7.9%)	70 (20.8%)

Tableau 3.16 – Sommaire des résultats de l'expérience 16

Les résultats de cette expérience montrent que WALA a été capable d'identifier correctement plus de nœuds et d'arcs que ACG sur le projet underscore, mais en contrepartie à générer plus de nœuds supplémentaires qui sont fort probablement des faux positifs (58 vs 468). 468 reste un nombre relativement faible si on considère qu'il y avait 337 arcs au total à identifier.

3.4 Expérimentations avec TAJS

3.4.1 Expérience 17 (TAJS - GhostCMS)

Pour cette expérience, nous avons exécuté TAJS sur le projet GhostCMS avec la même approche que l'expérience 7, 8 et 9. L'objectif de cette expérience est d'observer si TAJS rencontre les mêmes enjeux que SAFE.

NC	AC	NS	AS	NM	AM
0 (0%)	0 (0%)	5	2	3 958 (100%)	4 197 (100%)

TABLEAU 3.17 – Sommaire des résultats de l'expérience 17

Cette expérience montre que TAJS n'a pas été capable d'analyser le projet GhostCMS. TAJS a terminé son analyse prématurément et n'a rien été capable d'identifier.

CHAPITRE IV

ANALYSE DES RÉSULTATS ET DISCUSSION

4.1 Quelles sont les difficultés d'analyse ?

4.1.1 Identification des éléments problématiques

L'identification des éléments problématiques dans les approches analysées a été faite en utilisant comme point de départ les résultats de la section 3. Pour chaque expérience où les résultats étaient mauvais, un travail de fouille dans l'outil et le projet analysé a été fait pour identifier les causes racines des mauvais résultats. Les sections subséquentes présentes le sommaire de ce travail d'analyse.

4.1.2 Les arcs d'utilisation de bibliothèques

Un des constats qui peut être fait en analysant les arcs non trouvés par l'outil WALA est que certains arcs entre les fonctions de la bibliothèque underscore n'apparaissent qu'à l'utilisation. Ainsi, le code de la figure 4.1 va générer à l'exécution un arc entre la fonction `_.map` et `_.initial`. Comme le code de la figure 4.1 est uniquement présent dans les tests unitaires, l'analyse de WALA ne les identifie pas dans ces résultats.

FIGURE 4.1 – Utilisation de l'API _.map avec _.initial

```
_.map([[1, 2, 3], [1, 2, 3]], _.initial);
```

Ce cas met en lumière que l'analyse de code de bibliothèques faite sans tenir compte du contexte d'utilisation va nécessairement amener des imprécisions. Comme WALA ne fait aucune hypothèse sur ce qu'un client peut raisonnablement appeler, il n'arrive pas à identifier cet arc.

WALA pourrait, en théorie, faire des hypothèses sur ce qu'un client peut raisonnablement appeler en utilisant une approche comme Kristensen et Møller (2019). Ce type d'approche essaie de trouver un juste milieu entre surapproximer en considérant que tout peut être passé en argument et sous-approximer en considérant que rien ne peut être passé en argument. Par contre, ce type d'approche a nécessairement des imprécisions, étant donné que l'on fait des hypothèses.

4.1.3 Modélisation imparfaite de l'environnement d'exécution

Il y a une modélisation imparfaite de l'environnement d'exécution lorsqu'un outil ne tient pas compte de toutes les valeurs et tous les comportements provenant du contexte d'exécution dans lequel un programme est exécuté (ex. : variable globale `window`, variable globale `process`, etc.).

La modélisation imparfaite de l'environnement d'exécution amène toutes sortes de problèmes aux analyseurs. On pourrait être tenté d'affirmer que ces imperfections sont des détails d'implémentation et que les erreurs d'analyse qui en découlent ne sont pas importantes parce qu'elles sont corrigibles, mais, en pratique, ce n'est pas vraiment une analyse que l'on peut juger adéquate. Les environnements d'exécution JavaScript se sont multipliés au fil des années et on en retrouve une très

grande variété (node, deno, electron, WScript, etc.). Même au sein d'un même environnement comme "node", il y a des variations entre les versions. Le fait que la modélisation de l'environnement d'exécution est imparfaite fait donc partie des défis inhérents de l'analyse statique de code.

Pour l'analyseur SAFE, le principal problème que la modélisation imparfaite amène est que son analyse très précise a tendance à s'arrêter trop tôt, car il considère que toute valeur ou API non définie globalement n'existe pas et lance une exception à l'exécution. Lorsque l'on essaie d'analyser le projet GhostCMS avec SAFE l'analyseur termine son analyse très tôt, car il pense que la variable globale `process` n'existe pas et que le code va lancer une exception à l'exécution. Cette interprétation est fausse et fait en sorte que la quasi-totalité du code n'est pas analysée. Ce cas de figure montre qu'une analyse trop précise peut très mal performer dans un contexte d'environnement d'exécution variable ou inconnu. Un analyseur qui veut ainsi bien gérer cette incertitude doit avoir une approche qui a tendance à surapproximer face à des valeurs inconnues. On observe le même comportement dans l'outil TAJS.

Pour l'analyseur WALA, la modélisation imparfaite est la cause de la non-identification des arcs entre la fonction native `String.prototype.replace` et la fonction de `callback` passée en argument. Cette limitation n'est pas due à des difficultés d'analyse pour les fonctions passées en argument, puisque l'analyse de WALA est parfaitement capable de trouver les arcs entre la fonction native `sort` et les fonctions passées en argument. La source du problème est que la définition de `replace` dans le prologue [1] est définie comme retournant la deuxième valeur et ne tient pas compte du fait que le deuxième argument peut être une fonction. Le problème précis de la

1. Section de code JavaScript que WALA injecte au programme pour modéliser l'environnement d'exécution.

non-modélisation de `replace` a été corrigé dans la version 1.6.6 (commit e39d2d6) de WALA, mais le problème plus général que les erreurs de modélisation amène des arcs manquantes lui est toujours présent. Ce type de problème est fondamentalement difficile à adresser, parce qu'il nécessite soit une mise à jour constante de l'outil pour avoir une modélisation très précise de l'ensemble des environnements possibles ou que l'analyseur fasse des hypothèses qui amènent une surapproximation (ex. : assumer que lorsqu'une fonction est passée en argument à une fonction native, il va conservativement y avoir un arc entre la fonction native et la fonction passée en argument).

4.1.4 Tolérance aux erreurs d'implémentation

Un aspect intéressant qui ressort des analyses qui ont été faites est la capacité des approches à produire des résultats partiels en présence d'erreurs d'implémentation. Avec la complexité croissante de la spécification de JavaScript et des environnements qui l'exécute (ex. : NodeJS), il est ultimement difficile d'être toujours entièrement à jour sur toutes les fonctionnalités et d'avoir une modélisation parfaite. Il est donc intéressant à ce titre d'observer comment les approches s'en tirent quand ils rencontrent des obstacles dus à des fonctionnalités qu'ils ne supportent pas.

Dans le cas de ACG, quand l'analyse rencontre des fichiers avec des syntaxes non supportées (ex. : propriété privée [2]), ce qu'on observe est que soit le fichier au complet est ignoré ou que les nœuds liés à cette syntaxe sont ignorés. Tous les autres nœuds étaient eux correctement analysés et intégrés aux résultats. Ainsi, avant que l'on rajoute le support pour les nœuds "MethodDefinition", le compor-

2. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_elements`

tement de ACG était de ne pas inclure ces éléments comme des nœuds et tout le reste était inclus.

Dans le cas de SAFE, ce que l'on observe est que l'analyseur a tendance à terminer l'analyse de façon prématurée si des propriétés inattendues sont présentes ou qu'un comportement spécifique n'a pas été modélisé. Avant que l'on applique les correctifs discutés dans la section 2.9, SAFE terminait prématurément l'analyse parce qu'il pensait que `process` n'était pas une variable qui existait globalement. De plus, moindrement que l'outil n'était pas calibré correctement en termes de paramétrage, on entrait dans des conditions où une quantité exponentielle de mémoire ou de temps était nécessaire pour terminer l'analyse. C'est essentiellement le problème qui a été rencontré avec l'expérience 7 (SAFE avec paramètre par défaut - GhostCMS), 8 (SAFE maxStrSetSize - GhostCMS) et 9 (SAFE maxStrSetSize,loopIter,callsiteSensitivity - GhostCMS).

Quand on compare le comportement de ACG et SAFE, on peut observer que l'analyseur plus approximatif (ACG) a tendance à mieux performer par rapport à cet aspect que SAFE. Les erreurs dans le cas de ACG ont diminué la précision de l'analyse, mais pas de façon aussi significative que SAFE où l'analyseur ne sortait pratiquement aucun résultat sans correctif ou paramétrage important dans l'algorithme. Cet aspect montre la nécessité de ne pas avoir des analyses trop rigides pour bien performer avec des cas pratiques.

4.1.5 Propriétés dynamiques

Les propriétés dynamiques sont une difficulté pour les approches approximatives qui ne font pas une analyse complète du flux de données. La finalité du problème est que l'on arrive à un point dans l'analyse où l'information nécessaire pour analyser correctement la valeur n'a pas été propagée et l'analyse ne peut donc pas

trouver les arcs en question, puisqu'elle n'a pas l'information. Même si ce principe est plutôt simple à comprendre, il est tout de même intéressant de regarder les patrons qui créent des problèmes pour les analyseurs.

Patron de chemin dynamique

On voit, par exemple, dans l'approche par champ implémentée dans WALA [3] que le flux de données est uniquement fait entre les affectations directes et les propriétés de même nom lorsque le nom est statique. Cette approche est insuffisante pour analyser correctement la fonction `result` de la figure 4.2 que l'on retrouve dans la bibliothèque underscore. Dans cette fonction, le flux de données de WALA fait le lien entre la lecture de `obj` et le fait que c'est un argument de la fonction `result`, mais ne fait pas une analyse suffisamment précise pour extraire les valeurs possibles qui sont dérivées de `path` (ligne 11). Il ne peut donc pas savoir les valeurs possibles de `prop` qui en découlerait. Les arcs qui sont générés par ces appels ne sont donc pas trouvés par l'analyseur de WALA. Par exemple, l'appel `result({a : { b : function (){ /*...*/ }}}, "a.b")` contient un arc entre la fonction `result` et la fonction `function (){ /*...*/ }`, mais cet arc n'est pas identifié par WALA.

---

3. `https://github.com/wala/WALA/tree/master/cast/js/src/main/java/com/ibm/wala/cast/js/callgraph/fieldbased`

FIGURE 4.2 – Extrait de la bibliothèque underscore

```
1. // Traverses the children of ‘obj‘ along ‘path‘. If a child
    is a function, it
2. // is invoked with its parent as context. Returns the value
    of the final
3. // child, or ‘fallback‘ if any child is undefined.
4. function result(obj, path, fallback) {
5.   path = toPath(path);
6.   var length = path.length;
7.   if (!length) {
8.     return isFunction$1(fallback) ? fallback.call(obj) :
   fallback;
9.   }
10.  for (var i = 0; i < length; i++) {
11.    var prop = obj == null ? void 0 : obj[path[i]];
12.    if (prop === void 0) {
13.      prop = fallback;
14.      i = length; // Ensure we don't continue iterating.
15.    }
16.    obj = isFunction$1(prop) ? prop.call(obj) : prop;
17.  }
18.  return obj;
19. }
```

Une version simplifiée de ce qui pose problème pour l'approche par champ de WALA est le code de la figure 4.3. Ce qui pose problème à WALA dans cette version simplifiée est la résolution des valeurs possible de `parts[0]` et `parts[1]`. WALA ne peut donc pas trouver les arcs qui vont de la fonction "result" vers les fonctions "a" et "d".

FIGURE 4.3 – Reproduction simplifiée du problème présent dans la bibliothèque underscore

```
// Minimal case of dynamic called based on arguments
function a() {
  return { "d" : d };
}
function d() {
  return 1;
}
function result(obj, parts) {
  var a = obj[parts[0]]();
  var b = a[parts[1]]();
  return b;
}
console.log(result({ "e" : a }, ["e", "d"]));
```

Il est bon de noter que le patron d'accès de propriétés basés sur une valeur dynamique est présent dans d'autres bibliothèques que underscore. Dans la bibliothèque lodash, il y a, par exemple, la fonction `_.get` [4] qui offre une fonctionnalité similaire à la fonction `result` de underscore.

Ce type de code est difficile à analyser parce qu'il faut que l'analyse comporte une analyse de flux de données précise, sinon l'information nécessaire est manquante.

Patron d'itération d'objet

L'analyse basée sur les champs de WALA a les mêmes difficultés lorsqu'une fonction itère sur les propriétés d'un objet dynamique. Dans la bibliothèque underscore

4. `https://github.com/lodash/lodash/blob/4.17.15/lodash.js#L3028`

qui est analysée, la fonction `mixin` de la figure 4.4 utilise un patron (proxy) qui fait de l'itération d'objet. Lorsqu'un objet est passé à cette fonction, les fonctions contenues dans cet objet sont assignées à l'objet `_$1`. Le comportement attendu est que toutes les fonctions appelées sur l'objet `_$1` aient un arc de la fonction appelant vers la fonction interne qui appelle par la suite `chainResult`. WALA n'identifie pas ces arcs.

FIGURE 4.4 – Patron de proxy présent dans la bibliothèque underscore

```
function mixin(obj) {
  each(functions(obj), function(name) {
    var func = _$1[name] = obj[name];
    _$1.prototype[name] = function() {
      var args = [this._wrapped];
      push.apply(args, arguments);
      return chainResult(this, func.apply(_$1, args));
    };
  });
  return _$1;
}
```

4.1.6 Analyse précise de boucle

Faire l'analyse précise du comportement des boucles amène parfois des situations où l'analyse est trop coûteuse en termes d'espace mémoire ou de temps d'exécution pour être possible en pratique. Ce problème a été observé lors de l'analyse de la bibliothèque underscore par l'analyseur SAFE. Une version simplifiée du problème rencontré par SAFE pour l'analyse de underscore est le code de la figure 4.5. À chaque itération de la boucle, SAFE met à jour les valeurs possibles de `prev` avec le produit des valeurs courantes de `prev` et l'ensemble des éléments contenus dans

le tableau `arr1`. Le problème avec cette approche est qu'à la fin de la boucle, la variable `prev` va contenir 14^{14} valeurs possibles. Il est donc évident que l'analyseur va fondamentalement manquer de mémoire avec cette approche.

FIGURE 4.5 – Exemple simplifié d'analyse de code complexe avec des boucles

```
function a(s) {
    return s + "abc";
}

var arr1 = ["a0", "b0", "c0", "d0", "e0", "f0", "g0", "h0", "i0
   ", "j0", "k0", "l0", "m0", "n0"];
var prev = "";

for (var i=0; i<arr1.length; i++) {
    prev = prev + arr1[i];
    a(prev);
}
```

Ce problème avait déjà été mis en lumière dans l'article qui a introduit l'analyse précise de boucle pour JavaScript (Park et Ryu, 2015). La problématique plus générale qui avait été expliquée dans l'article est que chaque opération qui retourne un résultat indéterminé empêche l'analyse précise de la boucle de continuer avec des valeurs exactes et l'analyse doit continuer avec des valeurs inconnues qui amènent plus d'états à être considérés à chaque opération. À chaque étape où on doit considérer plus d'états, ces états se multiplient avec les états possibles précédents.

Ce problème montre que les analyseurs doivent borner le nombre de valeurs possibles qu'une variable peut avoir et/ou éviter de faire des produits d'état ou valeur possible. À défaut de prendre cela en considération, toute variante de cet exemple

de code aura un temps d'analyse exponentiel et un usage mémoire exponentiel. Cette observation est compatible avec l'état des connaissances globales du domaine.

L'autre aspect que ce problème met en lumière est que l'analyse précise de boucle est délicate à faire en pratique. Dans l'exemple donné, si on choisit de borner le nombre de valeurs possible et/ou de ne pas faire de produit croisé à chaque itération, la précision de notre analyse va être sensiblement la même que si on n'avait pas essayé de faire d'analyse précise de boucle. Cette observation amène à constater que l'analyse précise de boucle n'est pas nécessairement souhaitable à faire statiquement. Pour comprendre pourquoi l'analyse statique "pure" de boucle n'est pas nécessairement souhaitable, on peut creuser l'analyse un peu plus loin et regarder la question autant d'un point de vue théorique et pratique.

Analyse théorique

À très haut niveau, on peut séparer les boucles en deux catégories. Celles qui modifient l'état des variables locales qu'elles utilisent et celles qui ne le font pas. Pour les boucles qui modifient l'état des variables locales qu'elles utilisent, on a par définition quelque chose de la forme présentée à la figure 4.6.

FIGURE 4.6 – Boucle ayant un effet de bord à chaque itération

```
for (var i=0; i<n; i++) {
    local_var = expression(local_var, i)
}
```

Dans une boucle de cette forme, la complexité algorithmique d'exécution est au minimum $O(a^n)$ où `a` est le nombre de valeurs possible pour l'expression `expression(`

`local_var, i)` et `n` est le nombre maximum d'itérations de la boucle. La complexité exponentielle apparaît ici puisqu'à chaque itération on doit faire le produit des états de `local_var` et les nouvelles valeurs possibles de l'expression `expression( local_var, i)`.

La seule façon qu'il est possible d'analyser rapidement une boucle de façon précise est que le `n` soit très petit ou que le `a` soit 1. Obtenir de façon systématique un `a` de 1 implique que l'on est toujours capable d'estimer la valeur de l'expression et que notre estimation donne toujours une seule valeur possible. Pour le `n`, moindrement que le `a` à une valeur plus grande que 3 on obtient rapidement de grande valeur pour des `n` aussi petits que 5. Par exemple, avec une boucle qui introduit 4 états possibles et qui itère 5 fois, on obtient en sortie de boucle 1024 états possibles pour `local_var`. Chaque traitement successif de `local_var` devra prendre en compte ces 1024 états. Il est bon de noter ici que cette estimation se fait sur de très petite valeur (a=4 et n=5) et que l'on arrive déjà avec un nombre d'états important en sortie.

D'un point de vue théorique, il existe des approches qui ne créent pas d'explosion combinatoire telle que décrite dans le paragraphe précédent (ex. : une analyse par intervalle). Dans le cas des analyses par intervalle, même s'il y a une explosion des valeurs possibles, tant que ces nouvelles valeurs sont des valeurs dans l'intervalle existant, il n'y aura pas d'explosion combinatoire. Ceci vient du fait que la résultante est le même intervalle et donc on ne crée pas un nouvel état pour cet intervalle. L'enjeu avec cette approche est que les valeurs doivent se prêter à de l'analyse par intervalle (ex. : valeur numérique). Comme ce n'est pas quelque chose qui est généralisable à tous les types de valeurs possibles, les analyseurs utilisent peu ou pas ce type d'approche et font principale de l'analyse par ensemble. Pour cette raison, l'analyse présentée dans cette section porte uniquement sur l'analyse par ensemble.

L'autre approche possible pour ce cas est de ne pas analyser de façon précise et de faire de l'élargissement moindrement que le nombre d'états d'une variable dépasse un certain seuil. Si, par exemple, on voit qu'une variable a plus de 100 états, on remplace les états par $\top$. Cette approche garantit un temps d'exécution maximal pour les boucles et évite les problèmes de croissance exponentielle. C'est d'ailleurs l'idée originale des articles qui ont introduit le concept d'élargissement. La contrepartie de l'utilisation de l'élargissement avec l'analyse de boucle est que si on atteint trop rapidement le point où l'élargissement est appliqué, notre analyse de boucle ne modélisera que les premières itérations de la boucle. On est ainsi fondamentalement très proche d'une analyse où on ne considère pas les itérations des boucles. Comme l'objectif d'une analyse précise de boucles est d'obtenir plus d'informations qu'une analyse qui ne considère pas les itérations, on voit ici, que d'un point de vue théorique, le gain potentiel n'est pas complètement absent, mais qu'il existe clairement des cas où il ne sera pas possible d'avoir de bénéfices.

Les boucles qui ne modifient pas l'état des variables locales qu'ils utilisent ont par définition la forme de la figure 4.7.

FIGURE 4.7 – Boucle n'ayant pas d'effet de bord à chaque itération

```
for (var i=0; i<n; i++) {
    expression(local_var, i)
}
```

Dans une boucle de cette forme, `expression` est l'ensemble des instructions contenues dans la boucle, `n` est le nombre d'itérations de la boucle et `i` est la valeur courante de l'itération. L'analyse de ce type d'expression est à très haut niveau linéaire $O(n)$ où n est le nombre d'itérations de la boucle si l'analyse de l'expression `expression` est en temps constant. En principe, on devrait avoir ici un cas où

l'analyse précise de boucle est rapide et bénéfique. Par contre, lorsque l'on considère les alternatives à l'analyse précise de boucle, l'avantage de faire une analyse précise de boucle est plus flou. On pourrait, par exemple, analyser les expressions contenues dans les boucles en supposant que `i` à n'importe quelle valeur entre 0 et `n` et ignorer complètement le fait que c'est une boucle. Le seul cas où d'un point de vue théorique on aurait des gains pour l'analyse de boucle est lorsque l'on combine l'analyse de boucle avec le partitionnement de trace. Si on établit pour chaque itération une nouvelle partition, la variable `i` aura la même valeur pour chaque opération à l'intérieur de la boucle. Les article qui discutent d'analyse précise de boucle vont généralement utiliser le concept de partitionnement de trace pour cette raison (Park et Ryu, 2015).

Analyse pratique

Pour évaluer si l'intuition théorique expliquée dans "Analyse théorique" est valide en pratique, l'expérience suivante a été mise en place. Le point de départ de l'analyse est l'analyseur SAFE. Nous allons utiliser les configurations "maxStrSetSize" et "loopIter" qui sont applicables à l'étape de "HeapBuild" de l'outil. Ces configurations implémentent deux approches intéressantes à analyser. "maxStrSetSize" permet d'appliquer une approche d'élargissement lorsqu'une variable de type String contient trop de String différents possibles. Quand une variable dépasse le seuil indiqué, on considère que la variable peut contenir tous les chaînes de caractère possibles $\top$. "loopIter" permet de configurer si on veut une analyse de boucle ou non. Lorsqu'il n'y a pas d'analyse de boucle, on analyse le code comme s'il y avait une seule itération et quand il y a une analyse de boucle, on analyse chaque itération possible d'une boucle jusqu'à un nombre maximal d'itérations.

Chaque configuration de SAFE sera ensuite exécutée sur le projet underscore [5] et GhostCMS [6]. On va ensuite mesurer l'apport de l'analyse précise de boucle pour la construction du graphe d'appel en regardant la différence avec les résultats obtenus des différentes configurations de SAFE. Si l'analyse précise de boucle apporte peu, on devrait obtenir des résultats de précision similaires ou égaux entre les approches. Si l'analyse précise de boucle apporte quelque chose de significatif, on devrait observer que, pour l'approche utilisant l'élargissement ou l'approche sans modification, les résultats sont considérablement meilleurs.

Dans le cas de GhostCMS, il n'a pas été possible de trouver un paramétrage qui permet à l'analyse de terminer sans manquer de mémoire ou ne pas avoir terminé après plusieurs jours d'analyse. L'aspect de l'analyse de boucle n'a donc pas pu être analysé sur ce projet. Dans le cas de la bibliothèque underscore, pour que l'analyse puisse terminer dans un délai raisonnable, le paramètre "maxStrSetSize" a dû être défini à une valeur très petite ("10"). Avec ou sans "loopIter=1" les résultats étaient identiques (voir expérience 11 (SAFE maxStrSetSize,callsiteSensitivity - underscore) et expérience 12 (SAFE maxStrSetSize,callsiteSensitivity,loopIter - underscore)) en termes de nœuds et d'arcs identifiés. Il est bon de noter ici que le paramètre qui a été nécessaire pour l'analyse force un élargissement assez agressif. Si une valeur plus grande (ex. : 25) avait été choisie, l'analyse ne terminait pas après plusieurs jours.

Globalement, il est difficile avec ces résultats d'arriver à une réponse concluante par rapport à l'apport potentiel de l'analyse de boucle. Il a par contre été possible d'observer ici que les gains potentiels de l'analyse de boucle tendent à être nuls lorsque de l'élargissement est appliqué.

---

5. `https://underscorejs.org/`

6. `https://github.com/TryGhost/Ghost`

Il est intéressant de comparer cette analyse avec les résultats obtenus par l'article Park et Ryu (2015). Dans Park et Ryu (2015), l'évaluation a été faite en analysant des projets de petite et moyenne taille avec l'approche d'analyse de boucle qu'ils proposaient. Les résultats obtenus sont que leur approche apporte de meilleurs performances et résultats dans certains cas pour de petits et moyens projets. Par contre, les résultats indiquaient aussi que pour une portion des projets, leur approche souffrait quand même de problèmes liés à l'explosion du nombre d'états. Ce qu'on peut constater ici, c'est que plus un projet est gros plus la probabilité que du code problématique soit présent est grand. Donc, pour l'analyse de gros projets, les approches qui ont occasionnellement des cas limites problématiques sont peu utilisables, car la probabilité de la présence de ces cas limites est beaucoup plus grande que pour des petits projets.

4.1.7 Les fichiers superflus

Dans le cas de l'outil ACG, il a été observé que la présence importante de fichiers JavaScript superflus dans les dossiers de projet était problématique. Ces fichiers superflus sont principalement des fichiers JavaScript non utilisés qui sont distribués avec les bibliothèques utilisées par le projet analysé. Comme ACG ne fait pas d'analyse directe ou indirecte d'atteignabilité, tous les fichiers sont inclus à priori dans l'analyse. Le problème que ceci amène est qu'une bonne quantité de données non nécessaires doit être incluse dans l'analyse, ce qui affecte à la fois la performance et la précision des résultats. On observe dans les résultats de la construction du graphe d'appel des arcs qui vont vers des fichiers qui ne sont, en pratique, pas atteignables. On voit, par exemple, dans l'expérience 1 (ACG - GhostCMS sans node_modules) que ACG a déterminé qu'il y avait un arc entre le nœud 74001 (nœud du fichier frontend "frontend-caching.js") et le nœud 61766 (nœud du fichier backend "CustomThemeSettingsBREADService.js"). ACG faite

cette association, car le nom des fonctions est identique. Or, cet arc est impossible en pratique, parce que le backend ne peut pas appeler le frontend directement et vice-versa. Une analyse d'atteignabilité entre les fichiers aurait été suffisante pour montrer que le lien était impossible. Ce problème est particulièrement intéressant, parce qu'il est notoire que les projets JavaScript et particulièrement les projets NodeJS ont une quantité importante de fichiers superflus présents dans les dépendances. Au-delà de la précision des approches, l'analyse de fichiers superflus amène aussi des problèmes de performance majeurs. Par exemple, pour l'expérience 2 (ACG - GhostCMS sans les dossiers test/), après 2 jours d'exécution, l'analyse n'était toujours pas terminée. Ceci contraste beaucoup avec le faite que l'expérience 1 (ACG - GhostCMS sans node_modules) et 3 (ACG - GhostCMS avec fichiers de la ligne de base seulement) ont terminé dans un délai de l'ordre de grandeur de quelques minutes. On se trouve ainsi dans une position où, pour analyser de gros projets on doit ignorer des fichiers pertinents. Ceci affecte beaucoup la précision au final de l'analyse, car ignorer le dossier "node_modules" pour l'expérience 1 (ACG - GhostCMS sans node_modules) a fait en sorte que l'on a eu 3704 arcs manquants, alors que dans l'expérience 3 (ACG - GhostCMS avec fichiers de la ligne de base seulement) on a seulement eu 1591 arcs manquants et on est passé de 493 arcs correctement identifiés à 2606 entre les deux analyses. On peut donc constater qu'une grande partie de ce qui est manquant pour l'expérience 1 (ACG - GhostCMS sans node_modules) est due au fait que l'on a exclu "node_modules".

Ceci met en lumière l'importance de combiner une analyse d'atteignabilité à l'approche de ACG pour avoir des résultats plus précis et être plus performant. Cette idée est d'ailleurs quelque chose qui émerge d'articles plus récents sur le sujet comme "Indirection-Bounded Call Graph Analysis" (Chakraborty *et al.*, 2024).

4.1.8 Surapproximation et explosion combinatoire

Le fait que la surapproximation peut mener à une explosion combinatoire pour l'analyse de langages dynamiques est un problème connu dans le domaine. Une analyse trop conservatrice (où on considère trop d'éléments comme $\top$) a tendance à créer une analyse où il y a une explosion combinatoire, parce que trop de variables se retrouvent à avoir trop de valeurs possibles. On voit bien ce phénomène à l'œuvre dans les expériences 8 (SAFE maxStrSetSize - GhostCMS) et 9 (SAFE maxStrSetSize,loopIter,callsiteSensitivity - GhostCMS). Le patron qui cause ce problème est le suivant. Le transpilateur Babel a transpilé la création des classes avec un appel de fonction qui suit le patron "`_createClass([{key:"myFunction", value:function(){}}])`". Le problème qui survient par la suite est que l'ensemble des fonctions de classes se trouve déclaré et initialisé par la fonction `_createClass` interne de Babel. Comme SAFE surapproxime les valeurs possibles à l'intérieur des fonctions de Babel, l'analyse conclut qu'un appel de fonction sur un objet peut appeler n'importe quelle fonction de n'importe quelle classe. On se trouve donc à analyser une quantité excessive de chemins qui n'existent pas en pratique. Ceci explique le résultat de ces analyses où on observe que le code termine en erreur ou ne termine pas après plusieurs jours d'exécution.

4.1.9 L'aliasing

En JavaScript, une fonction est une valeur qui peut être affectée à des propriétés de n'importe quel nom. On parle d'aliasing lorsqu'on assigne une fonction à une propriété ou variable ayant un nom différent de l'original. Par exemple, dans le code `obj1.a = function(){}; obj2.b = obj1.a;`, la fonction est originalement déclarée sur un attribut de nom `a` et ensuite elle est affectée à une propriété de nom `b`.

L'aliasing n'est pas simple à bien gérer en pratique. Premièrement, l'aliasing peut se produire dynamiquement lorsque du code de la forme `objA[cleA] = objB[cleB]` est présent. Comme on ne connait pas, à priori, la valeur de `cleA` et `cleB`, on ne peut pas savoir l'aliasing se fait de quelle clé à quelle clé. Deuxièmement, il peut être hasardeux de gérer l'aliasing en regroupant les clés. Lorsque l'on regroupe ces clés, on les considère égales. Par exemple, pour le code `objA.cleA = objB.cleB`, les clés `cleA` et `cleB` font partie d'un même ensemble et sont considérées égales. Dans cet exemple, un appel à une fonction de nom `cleA` aurait comme cible toutes les fonctions de nom `cleA` et `cleB`. Ces regroupements sont généralement vrais uniquement pour une portion du code, mais pas pour l'ensemble du code. Cette approche peut ainsi autant trouver de vrais positifs qu'ajouter de faux positifs.

L'aliasing est un aspect fondamentalement problématique de ACG. Comme on établit la relation appelant vers appelé en se basant uniquement sur le nom, une même fonction qui est assignée à des propriétés de nom différent ne sera pas bien prise en compte. Ce que l'on a plus particulièrement voulu approfondir dans les résultats est si l'aliasing qui découle de l'usage de "import" ou "require" a causé des problèmes pour ACG.

D'un point de vue théorique, on peut considérer un projet avec quatre fichiers (figure 4.8, 4.9, 4.10 et 4.11) et que l'on a exécutés ACG sur ce projet. Dans le code d'exemple, les appels à la fonction `require` retournent l'objet qui a été exporté avec `export` dans le fichier cible. `require` retourne une valeur que l'on peut assigner à une variable de n'importe quel nom. Le résultat attendu est que l'on ait un arc entre l'appel de la variable `_a` vers la fonction du fichier a.js, un arc entre l'appel de la variable `b` vers la fonction du fichier b.js et un arc de l'appel à `foobar` vers la fonction du fichier c.js. On va constater que ACG fait uniquement les liens nécessaires lorsque le nom de la fonction est le même que le nom choisi pour l'importation (l'arc de l'appel de la variable `b` vers la fonction du fichier b.js)

ou lorsque ce qui est importé est un objet (l'arc de l'appel à `foobar` vers la fonction du fichier c.js). L'arc entre l'appel de la variable `_a` et la fonction du fichier a.js est manquante pour ACG.

FIGURE 4.8 – Fichier index.js

```
var _a = require('./a').default;
var b = require('./b').default;
var _c = require('./c').default;

_a();
b();
_c.foobar();
```

FIGURE 4.9 – Fichier a.js

```
export default function a() {};
```

FIGURE 4.10 – Fichier b.js

```
export default function b() {};
```

FIGURE 4.11 – Fichier c.js

```
export default { foobar: function() { } }
```

En pratique, quand on analyse les résultats de l'analyse de ACG sur le projet GhostCMS, on peut observer que l'on est presque exclusivement dans des cas similaire au cas de "c.js". Ainsi, même si l'importation de fichiers renomme des

valeurs, comme on importe principalement des objets et non des fonctions, ACG est quand même capable de faire adéquatement les liens nécessaires.

4.2 Questions de recherche

4.2.1 Q1 - Quelle précision peut-on s'attendre pour de l'analyse sur des programmes réels avec les approches existantes ?

Pour cette question de recherche, la réponse varie en fonction du type et de la grosseur du projet qui est analysé. Pour les petits programmes et les programmes sans aucune dépendance externe, les approches performent relativement bien dans la mesure où on observe des résultats comparables aux bons résultats publiés dans les articles qui ont introduit ces approches. Par exemple, lorsqu'on a exécuté ACG sur une des bibliothèques (pdf.js) qui avaient été utilisées dans l'article original de ACG (Feldthaus *et al.*, 2013), les résultats ont montré que ACG était capable de trouver 85% des arcs venant de l'analyse dynamique. Par contre, pour les projets plus complexes, plus gros et avec beaucoup de dépendances externes, les approches existantes ont pratiquement tous mal performé. On a observé beaucoup de cas où les approches en question n'étaient soit pas capable de terminer dans un délai raisonnable ou qu'il terminait avec des résultats trop partiel (la très grande majorité des nœuds et des arcs à identifier étaient manquants). Le meilleur résultat vient de ACG dans l'expérience 3 (ACG - GhostCMS avec fichiers de la ligne de base seulement) où 63% des arcs venant de la trace dynamique ont été identifiés. Le hic, c'est que cette expérience impliquait qu'une analyse d'atteignabilité a été préalablement faite. L'outil seul n'a donc pas atteint ce résultat. Les autres expériences avec ACG ont soit mené à une exécution qui ne terminait pas où à un résultat nettement inférieur (17% dans l'expérience 1).

Il est bon de noter que même si les approches existantes ont très mal performé

sur le gros projet (GhostCMS), certaines approches semblent avoir plus de potentiel d'amélioration que d'autres. Par exemple, les expériences semblent montrer qu'une analyse d'atteignabilité pourrait régler une bonne partie des problèmes rencontrés par ACG. Par contre, pour des approches comme SAFE, les expériences semblent montrer que l'approche de SAFE est trop fragile et dépendante de la paramétrisation pour être généralisable. Certains paramètres comme "maxStrSetSize" causent une explosion combinatoire s'il est configuré avec une valeur trop petite et ce même paramètre cause une explosion combinatoire pour différentes raisons si la valeur est configurée est trop grande. De plus, ce qui a été observé est qu'un ensemble de configuration peut bien fonctionner pour un projet à analyser (expérience 12 (SAFE maxStrSetSize,callsiteSensitivity,loopIter - underscore)) et être appliqué sur un autre projet (expérience 9 (SAFE maxStrSetSize,loopIter,callsiteSensitivity - GhostCMS)) sans que cela permette d'avoir une analyse qui termine.

4.2.2 Q2 - Quelles structures de programme sont difficiles à analyser ?

Ce qui est intéressant par rapport aux analyses qui ont été faites est que non seulement des structures ont pu être identifiées, mais le contexte et les patrons plus généraux qui font en sorte qu'ils sont présents et problématiques ont pu ressortir.

On note que les propriétés dynamiques sont sans surprise présentes dans les éléments qui ont été difficiles à analyser. Par rapport à cet aspect, il a aussi été possible d'identifier que le patron de "proxy" est un exemple d'usage où l'usage de propriétés dynamiques a été difficile à analyser en pratique. Comme la création d'objets "proxy" passe par des boucles qui ajoutent dynamiquement des propriétés, moindrement que l'analyseur en question n'était pas capable d'analyser ces

boucles et d'inférer le nom des propriétés, l'analyse était incapable de faire adéquatement les liens nécessaires avec cet objet. Le seul analyseur qui a été capable de court-circuiter ce problème est ACG, mais cela est principalement dû au fait que l'approche qu'il utilise n'essaie pas de comprendre le code à ce niveau.

On note aussi que la présence de fichiers superflus dans les projets est un enjeu pour certains analyseurs. Les analyseurs qui font explicitement ou implicitement une analyse d'atteignabilité comme SAFE n'ont pas eu d'enjeu à ce niveau. À ce titre, tous les approches par propagation ne devraient pas, en principe, avoir d'enjeu. Par contre, des analyseurs comme ACG qui ne font pas d'analyse d'atteignabilité ont plus de difficultés avec cet aspect. Dans les gros projets JavaScript, il a été observé qu'une grande partie des fichiers présents dans le dossier d'un projet sont superflus. Ces fichiers ont fait grossir la taille de l'analyse même s'ils ne sont ultimement jamais utilisés ou référencés. C'est donc une difficulté d'analyse à surmonter lorsqu'on prend une approche qui ne fait pas d'analyse d'atteignabilité.

La variété des environnements d'exécution JavaScript est un problème pour plusieurs analyseurs. Ce problème n'est pas inhérent au code lui-même, mais c'est quelque chose qui rend l'analyse difficile pour certains analyseurs. Les analyseurs qui essaient de faire une analyse trop précise (ex. : SAFE) ont tendance à diverger significativement de l'exécution réelle d'un programme moindrement que leur modélisation de l'environnement d'exécution n'est pas exacte. Ceci est un défi avec la variété des environnements d'exécution JavaScript qui existe en ce moment (ex. : Node, Deno, navigateur web, WScript, etc.). Pour que l'analyse faite par ces analyseurs soit correctement faite, ces analyseurs doivent soit être capables d'inférer les caractéristiques d'un environnement arbitraire par eux-mêmes ou qu'ils stockent l'état précalculé de tous les environnements possibles. Sans cet aspect, il n'est pas possible pour des analyseurs qui ont ce mode de fonctionnement de faire une analyse de fichier arbitraire correctement.

L'absence de contexte d'utilisation est un élément qui a été source d'erreur pour l'ensemble des analyseurs. Ceci n'est pas nécessairement un problème lié aux approches eux-mêmes, mais davantage un problème lié à la façon dont on analyse certains types de projets. Ainsi, lorsqu'on analyse un projet de type bibliothèque, certains arcs peuvent uniquement apparaître à l'utilisation. Ceci fait en sorte que, si l'on veut une analyse plus précise pour ce type de projet, on devrait tenir compte du contexte d'utilisation (ex. : le code de test) ou se baser sur ce qu'un client peut raisonnablement appeler.

4.2.3 Q3 - Pourquoi certaines approches sont précises ou performantes ? Pourquoi certains approches sont imprécises ou non performantes ?

Ce qui a été intéressant d'observer dans les analyses, c'est à la fois que la précision peut être obstacle à la performance, mais aussi que l'absence de précision pouvait aussi l'être. Il est plutôt évident qu'une analyse trop précise peut amener une explosion combinatoire, mais ce qui est moins évident et qui a été observé est que l'absence de précision peut aussi être un enjeu. Dans les langages dynamiques comme JavaScript, faire prématurément de l'élargissement a tendance à faire en sorte que l'on explore beaucoup de branches de code qui ne peuvent pas arriver en pratique. Ainsi, pour le code `obj[prop]()`, si on considère "prop" comme $\top$ on va explorer tous les attributs qui sont des fonctions, alors que, si on n'avait pas fait d'élargissement, on va seulement explorer les fonctions que "prop" peut réellement référencer. Ceci fait en sorte que les approches performantes vont généralement être celles qui sont capables de trouver le bon compromis de précision. On ne peut pas être dans l'un des deux extrêmes pour avoir quelque chose de performant. Ceci contraste avec la vision simpliste qu'on pourrait avoir de la relation entre la précision et la performance où plus une approche est imprécise plus elle est performante et à l'inverse plus une approche est précise plus elle est lente.

4.2.4 Q4 - Est-ce que la précision obtenue dans l'article Feldthaus *et al.* (2013) reste équivalente pour les très gros projets ?

Pour cette question, ce que l'on observe c'est que, pour les petits projets, on arrive à reproduire des résultats similaires à l'article original, mais pour les gros projets non. Lorsqu'il s'agit d'un programme relativement petit ou sans dépendance, la performance a tendance à être relativement bonne. La bibliothèque underscore et "pdf.js" répondent bien à ces critères et les résultats de précision étaient relativement bons au sens où ACG a été capable d'identifier la majorité des arcs et nœuds, il n'y avait pas trop de faux positif et l'ordre de grandeur du temps d'exécution était de quelques secondes à quelques minutes. Par contre, ACG n'a pas été capable d'obtenir de tel résultat pour l'analyse de GhostCMS. La présence de fichiers superflus a fait en sorte que l'on était soit dans un cas où l'analyse ne terminait pas ou que l'on devait couper toutes les dépendances pour que l'analyse termine dans un délai raisonnable. Aucun des deux choix n'était bon pour soit la performance ou la qualité des résultats. Ceci étant dit, les analyses ont aussi montré que si ACG intégrait un algorithme qui vérifierait l'atteignabilité des fichiers, les résultats obtenus seraient meilleurs. La qualité des résultats pour ACG pour les gros projets serait donc dépendante d'une analyse d'atteignabilité.

4.2.5 Q5 - Est-ce que l'analyse précise de boucle présentée dans Park et Ryu (2015) amène un gain potentiel de précision pour l'analyse des très gros projets ?

Pour cette question, les résultats ne sont pas entièrement concluants pour un "oui" ou un "non". Ce que les observations faites dans ce projet de recherche tendent à montrer est que les autres optimisations nécessaires pour avoir une analyse performante par propagation (comme ce que SAFE fait) ont tendance à amener des contraintes qui empêchent l'analyse précise de boucle de se faire et, par le fait

même, fait en sorte que l'analyse de boucle n'a aucun bénéfice. L'analyse précise de boucle n'est donc pas nécessairement en soit inutile, mais l'application de contrainte comme l'élargissement crée un contexte dans lequel il n'est pas possible pour cette approche d'avoir des bénéfices. Il serait donc intéressant dans des projets de recherche futurs d'explorer comment il serait possible de conserver les bénéfices potentiels de l'analyse de boucle lorsque d'autres contraintes importantes d'analyse sont appliquées.

CONCLUSION

5.1 Efficacité des nouveautés méthodologiques

Il y a deux nouvelles approches méthodologiques que ce projet de recherche a introduites et qui sont relativement nouvelles. Il est intéressant d'avoir un peu de rétrospection par rapport à leur efficacité à accomplir leurs objectifs.

Ce projet a tout d'abord présenté une approche agnostique des outils utilisés dans un projet pour obtenir des traces dynamiques d'exécution de projets. Précédemment, les outils publiés nécessitaient que les projets analysés utilisent "npm" ou que le point d'entrée des projets analysés s'exécute directement avec "node". Dans ce projet de recherche, il a été possible d'utiliser cette méthodologie pour obtenir des traces dynamiques de projet utilisant "npm test", "yarn test" et un script node directement. Le fait que la méthodologie ait pu marcher avec toute cette variété montre bien sa généricité, ce qui était l'objectif voulu. Le seul point négatif qui a été observé est le fait que le comportement de base de "yarn" n'interagit pas tout à fait correctement avec les scripts d'analyse dynamique, mais cet enjeu a pu être corrigé avec quelques lignes de "bash" spécifiques à "yarn" lorsqu'un projet utilise cet utilitaire. Aucun autre correctif spécifique à un outil n'a été nécessaire.

Ce projet a aussi présenté une approche pour transformer des projets structurés sur plusieurs fichiers en un seul fichier. Cette transformation avait pour objectif de permettre l'analyse de ces projets par des outils qui ne supportaient pas de façon inhérente un dossier de projet ou un ensemble de fichiers en entrée et d'être capable de le faire en minimisant la complexité d'analyse supplémentaire pour les

outils. L'atteinte de cet objectif s'est avérée plutôt mitigée. Là où l'approche a atteint ces objectifs est par rapport à être capable d'offrir une réécriture en un seul fichier qui est simple à analyser. Cet aspect de la réécriture n'a pas posé de problème d'analyse pour SAFE. Par contre, comme les plus vieux outils (ce sont principalement eux qui ne supportent pas de dossier ou ensemble de fichiers) ne supportent pas les syntaxes plus récentes de JavaScript, il a été nécessaire d'utiliser Babel pour transpiler le code combiné. Comme la transpilation de Babel a introduit des transformations qui sont difficiles à analyser, le résultat global a été que le projet transformé a été globalement plus difficile à analyser que nécessaire par SAFE. L'idée de base ne semble pas mauvaise, mais il faudrait l'étendre à davantage d'aspects du code JavaScript (pas juste le groupement de fichiers) pour être capable d'utiliser cette approche avec de plus vieux outils.

5.2 Contributions du mémoire

Globalement, dans ce projet de mémoire de maîtrise, les éléments suivants ont été des nouvelles contributions.

- L'outil ACG a été adapté pour être capable d'analyser des projets récents JavaScript.
- SAFE a été corrigé pour être fonctionnel dans un environnement contrôlé Docker.
- Une variante de SAFE est maintenant disponible pour analyser des projets dont l'environnement n'est pas bien modélisé par SAFE.
- Une approche pour obtenir des traces d'exécution dynamiques qui est un peu plus générique que ce qui décrit dans les articles de recherche.
- L'identification de plusieurs problématiques d'analyse que rencontrent les analyseurs statiques de code décrit dans des articles de recherche. Même

si ce travail de recherche ne propose pas de solution à ces problématiques, l'identification de celle-ci est une étape importante à leur résolution. D'autres recherches peuvent maintenant partir de ces problématiques pour proposer des solutions potentielles.

5.3 Menace à la validité

Certains aspects de ce projet de recherche pourraient mettre en cause la validité des conclusions et de la méthodologie. Le premier enjeu est le faible nombre de projets analysés. Il est possible que la méthodologie utilisée ait bien fonctionné pour le seul gros projet analysé (GhostCMS), mais ne fonctionne pas bien pour les autres gros projets JavaScript. Étant donné que le nombre de projets JavaScript analysés est petit, il est possible que les enjeux soulevés soient très spécifiques aux projets analysés et ne soient pas généralisables à davantage de projets JavaScript. Le deuxième enjeu est que la source de vérité utilisée pour la comparaison des résultats n'est pas garantie d'être complète. Il est possible que le nombre réel de vrais positifs soit plus grand que ce qui a été rapporté dans les résultats des expérimentations.

5.4 Questions ouvertes

Ce projet de recherche a mis en lumière les difficultés que les analyseurs ont présentement à faire de l'analyse statique sur des fichiers et projets JavaScript. Ces difficultés sont aussi des opportunités de recherche dans le sens où l'amélioration de ces aspects améliorerait la précision ou la performance des analyseurs existants.

Il est aussi intéressant de noter que certaines analyses ont mis en lumière que certaines avenues avaient plus de potentiel d'avancement que d'autres. Malgré sa très grande simplicité, ACG semble être l'approche qui a le plus de potentiel pour

l'analyse des grands projets. Le plus gros enjeu par rapport à ACG était l'absence d'analyse d'atteignabilité et cet aspect n'est pas en principe difficile à mettre en place. Il serait donc intéressant de voir davantage de recherche faite sur ACG dans laquelle on intègre des analyses d'atteignabilité.

5.5 Perspectives

L'objectif initial de ce projet de recherche était d'améliorer les outils existants d'analyse statique de code JavaScript pour qu'ils soient mieux adaptés aux gros projets. Même si ce projet de recherche n'a pas mis de l'avant de nouvelles approches, l'amélioration des connaissances générales par rapport à ce qui fonctionne bien ou pas en pratique est une avancée non négligeable. Avec ces connaissances, il est possible de mieux orienter la recherche et le développement d'outils en préconisant les approches qui sont davantage adaptées aux gros projets. On espère ainsi voir ces résultats de recherche être utilisés à cette fin.

RÉFÉRENCES

Anders, Holm, T. P. J. S. et Møller (2009). Type analysis for javascript. Dans Z. P. Jens et Su (dir.). *Static Analysis*, 238–255. Springer Berlin Heidelberg.

Anderson, C., Giannini, P. et Drossopoulou, S. (2005). Towards type inference for javascript. Dans *Proceedings of the 19th European Conference on Object-Oriented Programming*, 428–452. Springer-Verlag. http://dx.doi.org/10.1007/11531142_19

Andreasen, E. et Møller, A. (2014). Determinacy in static analysis for jquery. *SIGPLAN Not.*, *49*(10), 17–31. http://dx.doi.org/10.1145/2714064.2660214

Andreasen, E. S., Møller, A. et Nielsen, B. B. (2017). Systematic approaches for increasing soundness and precision of static analyzers. Dans *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 31–36. Association for Computing Machinery. http://dx.doi.org/10.1145/3088515.3088521

Antal, G., Hegedűs, P., Herczeg, Z., Lóki, G. et Ferenc, R. (2023). Is javascript call graph extraction solved yet ? a comparative study of static and dynamic tools. *IEEE Access*, *11*, 25266–25284. http://dx.doi.org/10.1109/ACCESS.2023.3255984

Arteau, O. (2018). *Prototype pollution attack in NodeJS application, https://raw.githubusercontent.com/HoLyVieR/prototype-pollution-nsec18/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf*, Dans *NorthSec*. Olivier Arteau.

Bandhakavi, S., Tiku, N., Pittman, W., King, S. T., Madhusudan, P. et Winslett, M. (2011). Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, *54*, 91–99. http://dx.doi.org/10.1145/1995376.1995398

Chakraborty, M., Gnanakumar, A., Sridharan, M. et Møller, A. (2024). Indirection-bounded call graph analysis. Dans J. Aldrich et G. Salvaneschi (dir.). *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313, 10 :1–10 :22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Keywords : JavaScript, call graphs, points-to analysis, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2024.10

Chugh, R., Meister, J. A., Jhala, R. et Lerner, S. (2009). Staged information flow for javascript. Dans *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 50–62. Association for Computing Machinery. http://dx.doi.org/10.1145/1542476.1542483

Cousot, P. et Cousot, R. (1977). Static determination of dynamic properties of recursive procedures. Dans E. Neuhold (dir.). *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, 237–277. North-Holland.

Fass, A., Somé, D. F., Backes, M. et Stock, B. (2021). Doublex : Statically detecting vulnerable data flows in browser extensions at scale. Dans *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 1789–1804. Association for Computing Machinery. http://dx.doi.org/10.1145/3460120.3484745

Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J. et Tip, F. (2013). Efficient construction of approximate call graphs for javascript ide services. Dans *2013 35th International Conference on Software Engineering (ICSE)*, 752–761. http://dx.doi.org/10.1109/ICSE.2013.6606621

Guarnieri, S. et Livshits, B. (2009). Gatekeeper : mostly static enforcement of security and reliability policies for javascript code. Dans *Proceedings of the 18th Conference on USENIX Security Symposium*, 151–168. USENIX Association.

Guarnieri, S. et Livshits, B. (2010). Gulfstream : staged static analysis for streaming javascript applications. Dans *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, p. 6., USA. USENIX Association.

Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S. et Berg, R. (2011). Saving the world wide web from vulnerable javascript. Dans *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 177–187. Association for Computing Machinery. `http://dx.doi.org/10.1145/2001420.2001442`

Jang, D. et Choe, K.-M. (2009). Points-to analysis for javascript. Dans *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, 1930–1937., New York, NY, USA. Association for Computing Machinery. `http://dx.doi.org/10.1145/1529282.1529711`

Jensen, S. H., Jonsson, P. A. et Møller, A. (2012). Remedying the eval that men do. Dans *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 34–44. Association for Computing Machinery. `http://dx.doi.org/10.1145/2338965.2336758`

Julian, Satish, C., Max, S., Manu, T. F. S. et Dolby (2012). Correlation tracking for points-to analysis of javascript. Dans J. Noble (dir.). *ECOOP 2012 - Object-Oriented Programming*, 435–458. Springer Berlin Heidelberg.

Kashyap, V., Dewey, K., Kuefner, E. A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B. et Hardekopf, B. (2014). Jsai : a static analysis platform

for javascript. Dans *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 121–132. Association for Computing Machinery. http://dx.doi.org/10.1145/2635868.2635904

Kashyap, V., Sarracino, J., Wagner, J., Wiedermann, B. et Hardekopf, B. (2013). Type refinement for static analysis of javascript. *SIGPLAN Not.*, *49*(2), 17–26. http://dx.doi.org/10.1145/2578856.2508175

Ko, Y., Lee, H., Dolby, J. et Ryu, S. (2015). Practically tunable static analysis framework for large-scale javascript applications (t). Dans *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 541–551. http://dx.doi.org/10.1109/ASE.2015.28

Kristensen, E. K. et Møller, A. (2019). Reasonably-most-general clients for javascript library analysis. Dans *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 83–93. IEEE Press. http://dx.doi.org/10.1109/ICSE.2019.00026. Récupéré de https://doi.org/10.1109/ICSE.2019.00026

Lee, C. et Son, S. (2023). Adcpg : Classifying javascript code property graphs with explanations for ad and tracker blocking. Dans *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 3505–3518. Association for Computing Machinery. http://dx.doi.org/10.1145/3576915.3623084

Lee, H., Won, S., Jin, J., Cho, J. et Ryu, S. (2012). Safe : Formal specification and implementation of a scalable analysis framework for ecmascript. Récupéré de https://api.semanticscholar.org/CorpusID:14633517

Li, S., Kang, M., Hou, J. et Cao, Y. (2021). Detecting node.js prototype pollution vulnerabilities via object lookup analysis. Dans *Proceedings of the 29th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 268–279. Association for Computing Machinery. http://dx.doi.org/10.1145/3468264.3468542

Li, S., Kang, M., Hou, J. et Cao, Y. (2022). Mining node.js vulnerabilities via object dependence graph and query. Dans *31st USENIX Security Symposium (USENIX Security 22)*, 143–160., Boston, MA. USENIX Association.

Madsen, M., Livshits, B. et Fanning, M. (2013). Practical static analysis of javascript applications in the presence of frameworks and libraries. Dans *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 499–509., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/2491411.2491417

Madsen, M., Tip, F. et Lhoták, O. (2015). Static analysis of event-driven node.js javascript applications. Dans *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 505–519. Association for Computing Machinery. http://dx.doi.org/10.1145/2814270.2814272

Meawad, F., Richards, G., Morandat, F. et Vitek, J. (2012). Eval begone ! semi-automated removal of eval from javascript programs. Dans *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 607–620. Association for Computing Machinery. http://dx.doi.org/10.1145/2384616.2384660

Nielsen, B. B., Hassanshahi, B. et Gauthier, F. (2019). Nodest : feedback-driven static analysis of node.js applications. Dans *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 455–465. Association for Computing Machinery. http://dx.doi.org/10.1145/3338906.3338933

Nielsen, B. B. et Møller, A. (2020). Value partitioning : A lightweight approach to relational static analysis for javascript. Dans *Proc. 34th European Conference on Object-Oriented Programming (ECOOP)*.

Nielsen, B. B., Torp, M. T. et Møller, A. (2021). Modular call graph construction for security scanning of node.js applications. Dans *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, 29–41., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/3460319.3464836

Nilsson-Nyman, E., Hedin, G., Magnusson, E. et Ekman, T. (2009). Declarative intraprocedural flow analysis of java source code. *Electronic Notes in Theoretical Computer Science*, *238*, 155–171. Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008), http://dx.doi.org/https://doi.org/10.1016/j.entcs.2009.09.046

Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A. et Saxena, P. (2015). Auto-patching dom-based xss at scale. Dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 272–283. Association for Computing Machinery. http://dx.doi.org/10.1145/2786805.2786821

Park, C. et Ryu, S. (2015). Scalable and precise static analysis of javascript applications via loop-sensitivity. Dans J. T. Boyland (dir.). *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37, 735–756. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Keywords : JavaScript, static analysis, loops, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.735

Park, C., Won, S., Jin, J. et Ryu, S. (2015). Static analysis of javascript web applications in the wild via practical dom modeling (t). Dans *2015 30th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), 552–562. http://dx.doi.org/10.1109/ASE.2015.27

Park, J., Lim, I. et Ryu, S. (2016). Battles with false positives in static analysis of javascript web applications in the wild. Dans *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, 61–70., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/2889160.2889227

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, *74*, 358–366. Récupéré de https://api.semanticscholar.org/CorpusID:120980829

Rival, X. et Mauborgne, L. (2007). The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, *29*, 26–es. http://dx.doi.org/10.1145/1275497.1275501

Santos, J. F., Maksimović, P., Grohens, T., Dolby, J. et Gardner, P. (2018). Symbolic execution for javascript. Dans *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/3236950.3236956. Récupéré de https://doi.org/10.1145/3236950.3236956

Santos, J. F., Maksimović, P., Sampaio, G. et Gardner, P. (2019). Javert 2.0 : compositional symbolic execution for javascript. *Proc. ACM Program. Lang.*, *3*. http://dx.doi.org/10.1145/3290379

Seifert, D., Wan, M., Hsu, J. et Yeh, B. (2022). An asynchronous call graph for javascript. Dans *2022 IEEE/ACM 44th International Conference on Software*

Engineering : Software Engineering in Practice (ICSE-SEIP), 29–30. http://dx.doi.org/10.1145/3510457.3513059

Sen, K., Kalasapur, S., Brutch, T. et Gibbs, S. (2013). Jalangi : a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. Dans *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 615–618., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/2491411.2494598

Shivers, O. G. (1991). *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University.

Shiyi, B. G. W. et Ryder (2014). State-sensitive points-to analysis for the dynamic behavior of javascript objects. Dans R. Jones (dir.). *ECOOP 2014 - Object-Oriented Programming*, 1–26. Springer Berlin Heidelberg.

Taly, A., Erlingsson, U., Mitchell, J. C., Miller, M. S. et Nagra, J. (2011). Automated analysis of security-critical javascript apis. Dans *2011 IEEE Symposium on Security and Privacy*, 363–378. http://dx.doi.org/10.1109/SP.2011.39

Vitek, J., Horspool, R. N. et Uhl, J. S. (1992). Compile-time analysis of object-oriented programs. Dans *Proceedings of the 4th International Conference on Compiler Construction*, CC '92, p. 236–250., Berlin, Heidelberg. Springer-Verlag.

Wei, S. et Ryder, B. G. (2015). Adaptive Context-sensitive Analysis for JavaScript. Dans J. T. Boyland (dir.). *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 de *Leibniz International Proceedings in Informatics (LIPIcs)*, 712–734., Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.712

Welearegai, G. B., Schlueter, M. et Hammer, C. (2019). Static security evaluation of an industrial web application. Dans *Proceedings of the 34th ACM/-SIGAPP Symposium on Applied Computing*, SAC '19, 1952–1961., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/3297280.3297471

Yamaguchi, F., Golde, N., Arp, D. et Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. Dans *2014 IEEE Symposium on Security and Privacy*, 590–604. http://dx.doi.org/10.1109/SP.2014.44

Younang, A. et Lu, L. (2015). Improving precision of javascript program analysis with an extended domain of intervals. Dans *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, 441–446. http://dx.doi.org/10.1109/COMPSAC.2015.175

Zhang, M. et Meng, W. (2020). Detecting and understanding javascript global identifier conflicts on the web. Dans *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 38–49. Association for Computing Machinery. http://dx.doi.org/10.1145/3368089.3409747