



SECURING EVERY BIT OF YOUR DATA

# État de l'art : Techniques de fuzzing, exécution symbolique, slicing et combinaisons

---

Poste #1 : Rapport d'étude PASTIS

11 Janvier 2020  
Version : 1.3

Ref: 18-11-488-REP

## Contacts

<i>Quarkslab :</i>	Jean-Philippe Luyten	Chef de projet
	Robin David	Ingénieur R&D Sécurité Logicielle
	Paul Hernault	Ingénieur R&D Sécurité Logicielle
	Boyan Milanov	Ingénieur R&D Sécurité Logicielle
	Christian Heitman	Ingénieur R&D Sécurité Logicielle
	Jonathan Salwan	Ingénieur R&D Sécurité Logicielle
<i>DGA-MI :</i>	Justin Bourroux	Ingénieur DGA-MI

---

Quarkslab SAS – 13 rue Saint-Ambroise – 75011 Paris – France  
Tel : +33 (0)1 58 30 81 51

**Quarkslab**  
SECURING EVERY BIT OF YOUR DATA



*Ce rapport a été financé par la Direction Générale de l'Armement et réalisé dans le cadre du projet PASTIS.*



---

# Table des matières

---

<b>Part I Introduction</b>	<b>1</b>
<b>Chapitre 1 Information sur le projet</b>	<b>3</b>
1.1 Contacts . . . . .	3
<b>Chapitre 2 Synthèse des résultats</b>	<b>5</b>
2.1 Contexte . . . . .	5
2.2 Plan du rapport . . . . .	6
2.3 Synthèse de l'étude . . . . .	6
2.4 Synthèse des propositions de combinaisons . . . . .	8
<b>Chapitre 3 Introduction</b>	<b>9</b>
3.1 Analyses automatisées de programmes . . . . .	9
3.1.1 Aperçu des techniques d'analyse existantes . . . . .	9
3.1.2 Techniques d'analyse pour Programme d'Analyse Statique et de Tests Instrumentés pour la Sécurité (PASTIS) . . . . .	11
3.2 Questions de recherche . . . . .	11
3.3 Définitions de vulnérabilité . . . . .	12
3.4 Sélection des documents . . . . .	12
3.5 Méthodologie d'analyse . . . . .	13

<b>Part II Base de tests</b>	<b>15</b>
<b>Chapitre 4 Etat de l'art : Base de tests</b>	<b>17</b>
4.1 Introduction . . . . .	17
4.2 Etat de l'art des bases existantes . . . . .	18
4.3 Génération automatique de bugs . . . . .	22
4.4 Choix de la base de tests . . . . .	24
<b>Chapitre 5 Base de tests</b>	<b>25</b>
5.1 Notations . . . . .	25
5.2 Critères d'évaluation . . . . .	25
5.3 Tests atomiques . . . . .	27
5.3.1 UT_1 : Calcul de prédicat de chemin et modélisation mémoire .	29
5.3.2 UT_2 : Symbolisation des entrées et modélisation des contraintes	30
5.3.3 UT_3 : Exploration du programme . . . . .	33
5.3.4 UT_4 : Découverte de bugs . . . . .	35
5.4 Test de passage à l'échelle . . . . .	35
5.5 Conditions & Environnement d'Exécution . . . . .	36
5.5.1 Méthodologie & Protocole experimental de test . . . . .	36
5.5.2 Environnement de compilation . . . . .	38
5.5.3 Reproductibilité . . . . .	39
<b>Part III Exécution Concolique</b>	<b>43</b>
<b>Chapitre 6 État de l'art : Exécution Symbolique Dynamique</b>	<b>45</b>
6.1 Introduction . . . . .	45

---

6.2	Définitions & Algorithme . . . . .	46
6.2.1	Prédicat de chemin . . . . .	46
6.2.2	Algorithme d'exécution symbolique . . . . .	46
6.3	Représentation intermédiaire . . . . .	47
6.3.1	Introduction . . . . .	47
6.3.2	Représentations intermédiaires existantes . . . . .	48
6.3.3	Lifter & Transducteurs d'IR . . . . .	51
6.3.4	Correction des IRs . . . . .	52
6.3.5	Comparatifs entre IRs . . . . .	52
6.3.6	Traitements de l'IR . . . . .	52
6.4	Exécution symbolique dynamique . . . . .	53
6.4.1	Introduction . . . . .	53
6.4.2	Concrétisations & Symbolisations . . . . .	53
6.4.3	Sources de valeurs concrètes & Instrumentation dynamique . . .	54
6.4.4	Calcul du prédicat d'un chemin . . . . .	55
6.4.5	Optimisations . . . . .	56
6.4.6	Requête sur un chemin & Exploitabilité . . . . .	56
6.5	Modélisation de la mémoire . . . . .	56
6.5.1	Introduction . . . . .	56
6.5.2	Stratégies de couverture des valeurs de pointeurs . . . . .	57
6.5.3	Procédures de décision . . . . .	58
6.6	Gestion des entrées et modélisations symboliques . . . . .	59
6.6.1	Problématique . . . . .	59
6.6.2	Gestion des appels de bibliothèques . . . . .	60
6.6.3	Gestion des appels système . . . . .	60

## *Table des matières*

---

6.6.4	Gestion des instructions non déterministes . . . . .	60
6.7	Couverture des chemins . . . . .	61
6.7.1	Introduction . . . . .	61
6.7.2	Stratégies de couverture . . . . .	62
6.7.3	Calcul en arrière . . . . .	63
6.7.4	Summarisation . . . . .	64
6.7.5	Fusion d'états . . . . .	65
6.8	Résolution des formules . . . . .	66
6.8.1	Introduction . . . . .	66
6.8.2	Mode interactif . . . . .	67
6.8.3	Théories . . . . .	67
6.8.4	Outils de résolution de contraintes . . . . .	68
6.8.5	Optimisation de formules . . . . .	69
6.9	Analyse semi-automatisée, annotations manuelles . . . . .	71
6.10	Outils d'exécution Symbolique . . . . .	72
6.11	État de l'art : Conclusion intermédiaire . . . . .	72
<b>Chapitre 7 Analyse détaillée des outils</b>		<b>77</b>
7.1	Outils de DSE sélectionnés . . . . .	77
7.2	DSE #1 : Mayhem . . . . .	80
7.2.1	Contributions . . . . .	80
7.2.2	Considérations de conception . . . . .	80
7.2.3	Déroulement d'une analyse . . . . .	82
7.2.4	Exécution concolique “hybride” . . . . .	83
7.2.5	Modèle mémoire basé sur les index . . . . .	83

---

7.2.6	Conclusion . . . . .	84
7.3	DSE #2 : <b>Manticore</b> . . . . .	86
7.3.1	Caractéristiques fonctionnelles . . . . .	86
7.3.2	Stratégies de couverture . . . . .	87
7.3.3	Gestion de la mémoire . . . . .	87
7.3.4	Modélisation du système et des entrées . . . . .	88
7.3.5	Résolution des formules . . . . .	89
7.3.6	Exécution des tests . . . . .	90
7.3.7	Conclusion . . . . .	92
7.4	DSE #3 : <b>KLEE</b> . . . . .	94
7.4.1	Fonctions intrinsèques . . . . .	94
7.4.2	Gestion des entrées symboliques . . . . .	95
7.4.3	Gestion de la mémoire . . . . .	95
7.4.4	Gestion des stratégies de recherche . . . . .	95
7.4.5	Résolution des formules & Optimisations . . . . .	96
7.4.6	Compilation et utilisation . . . . .	97
7.4.7	Exécution des benchmarks LAVA-M . . . . .	98
7.4.8	Suite de tests . . . . .	99
7.4.9	Conclusion . . . . .	99
7.5	DSE #4 : <b>Angr</b> . . . . .	100
7.5.1	Fonctionnalités & Écosystème . . . . .	100
7.5.2	Représentation intermédiaire . . . . .	101
7.5.3	Gestion des états . . . . .	102
7.5.4	Exécution & Stratégies d'exploration . . . . .	103
7.5.5	Modélisation des fonctions . . . . .	103

## *Table des matières*

---

7.5.6	Modélisation de la mémoire . . . . .	104
7.5.7	Symbion : Fusion de l'exécution concrète et de l'exécution symbolique . . . . .	104
7.5.8	Résolution de formules . . . . .	105
7.5.9	Utilisation . . . . .	105
7.5.10	Conclusion . . . . .	107
7.6	DSE #5 : Triton . . . . .	108
7.6.1	Architecture et fonctionnement . . . . .	109
7.6.2	Résolution des formules et gestion de la mémoire . . . . .	111
7.6.3	Propagation de la teinte . . . . .	111
7.6.4	Couplage avec une instrumentation dynamique . . . . .	111
7.6.5	Stratégies de couverture . . . . .	112
7.6.6	Utilisation . . . . .	112
7.6.7	Gestion des appels de bibliothèques . . . . .	113
7.6.8	Conclusion . . . . .	115
<b>Chapitre 8</b>	<b>Évaluation des outils</b>	<b>117</b>
8.1	Bombes logiques : Évaluation & Analyse des résultats . . . . .	117
8.1.1	Résultats de UT_1 . . . . .	117
8.1.2	Résultats de UT_2 . . . . .	119
8.1.3	Résultats de UT_3 . . . . .	119
8.1.4	Résultats de UT_4 . . . . .	120
8.1.5	Cas de tests . . . . .	121
8.2	LAVA : Évaluation & Analyse des résultats . . . . .	122
8.3	Synthèse & Conclusion des benchmarks . . . . .	122
8.3.1	Synthèse des tests atomiques . . . . .	122

---

8.3.2	Synthèse des tests de passage à l'échelle . . . . .	124
8.3.3	Synthèse sur les outils de DSE . . . . .	124
<b>Part IV</b>	<b>Fuzzing</b>	<b>127</b>
	<b>Chapitre 9 État de l'art : Fuzzing</b>	<b>129</b>
9.1	Introduction . . . . .	129
9.2	Algorithme . . . . .	130
9.3	Preprocessing . . . . .	132
9.3.1	Instrumentation . . . . .	132
9.3.2	Construction de l'ensemble de configurations initiales . . . . .	133
9.4	Ordonnancement . . . . .	133
9.5	Génération des entrées . . . . .	134
9.5.1	Génération par modèle . . . . .	135
9.5.2	Génération par mutation . . . . .	137
9.6	Exécution du programme . . . . .	138
9.6.1	Optimisations . . . . .	138
9.6.2	Oracles de bugs . . . . .	139
9.6.3	Triage des bugs . . . . .	141
9.7	Mise à jour des configurations . . . . .	143
9.7.1	Sélection des graines . . . . .	143
9.7.2	Mise à jour des graines . . . . .	143
9.7.3	Boucle de rétrocontrôle . . . . .	144
9.8	Fuzzing de protocoles réseau . . . . .	144
9.8.1	Problématique de l'oracle . . . . .	145

## Table des matières

---

9.8.2	Problématique des modèles . . . . .	145
9.8.3	Problématique de la machine à états . . . . .	145
9.8.4	Approche en <i>grey-box</i> . . . . .	147
9.8.5	Conclusion . . . . .	148
9.9	Outils de fuzzing . . . . .	148
9.10	Plateformes et infrastructures de fuzzing . . . . .	148
9.11	État de l'art : Conclusion intermédiaire . . . . .	150
<b>Chapitre 10 Analyse détaillée des outils</b>		<b>153</b>
10.1	Outils de Fuzzing sélectionnés . . . . .	153
10.2	Fuzzing #1 : American Fuzzy Lop (AFL) . . . . .	155
10.2.1	Prétraitement . . . . .	156
10.2.2	Gestion des entrées . . . . .	158
10.2.3	Exécution de la cible . . . . .	159
10.2.4	Conclusion . . . . .	161
10.3	Fuzzing #2 : Honggfuzz . . . . .	163
10.3.1	Prétraitement . . . . .	164
10.3.2	Gestion des entrées . . . . .	165
10.3.3	Exécution de la cible . . . . .	166
10.3.4	Gestion des crashes . . . . .	168
10.3.5	Conclusion . . . . .	168
10.4	Fuzzing #3 : AFL/QBDI . . . . .	170
10.4.1	Exécution de la cible . . . . .	170
10.4.2	Optimisations . . . . .	170
10.4.3	Conclusion . . . . .	173

---

10.5 Fuzzing #4 : PULSAR . . . . .	174
10.5.1 Description de l'outil . . . . .	174
10.5.2 Cas d'études . . . . .	177
10.5.3 Conclusion . . . . .	177
<b>Chapitre 11 Évaluation des outils</b>	<b>179</b>
11.1 Bombes logiques . . . . .	179
11.2 LAVA . . . . .	180
11.2.1 AFL et AFL/QBDI . . . . .	180
11.2.2 Honggfuzz . . . . .	183
11.3 Synthèse & Conclusion . . . . .	185
<b>Part V Combinaisons d'analyses</b>	<b>187</b>
<b>Chapitre 12 État de l'art : Combinaisons d'analyses</b>	<b>189</b>
12.1 Introduction . . . . .	189
12.2 Problématiques . . . . .	190
12.3 Fuzzers Whitebox . . . . .	190
12.3.1 Fuzzing guidé . . . . .	191
12.4 Cyber-Reasoning-System . . . . .	191
12.5 Analyse compositionnelle . . . . .	194
12.6 Etat de l'art : Conclusion intermédiaire . . . . .	194
<b>Chapitre 13 Analyse détaillée des outils</b>	<b>197</b>
13.1 Outils de combinaisons sélectionnés . . . . .	197
13.2 Combinaison #1 : Driller . . . . .	199

## *Table des matières*

---

13.2.1	Introduction . . . . .	199
13.2.2	Travaux connexes . . . . .	199
13.2.3	Fonctionnement . . . . .	200
13.2.4	Implémentation . . . . .	201
13.2.5	Conclusion . . . . .	201
13.3	Combinaison #2 : <i>Angora</i> . . . . .	202
13.3.1	Introduction . . . . .	202
13.3.2	Fonctionnalités . . . . .	203
13.3.3	Instrumentation . . . . .	206
13.3.4	Optimisations . . . . .	206
13.3.5	Conclusion . . . . .	207
13.4	Combinaison #3 : <i>Qsym</i> . . . . .	208
13.4.1	Problématique . . . . .	208
13.4.2	Approche . . . . .	209
13.4.3	Optimisations . . . . .	209
13.4.4	Implémentation . . . . .	211
13.4.5	Conclusion . . . . .	211
13.5	Combinaison #4 : <i>cyberdyne</i> . . . . .	213
13.5.1	Architecture . . . . .	213
13.5.2	Fuzzing . . . . .	214
13.5.3	Exécution symbolique . . . . .	215
13.5.4	Minset . . . . .	215
13.5.5	Conclusion . . . . .	216
13.6	Combinaison #5 : <i>Eclipser</i> . . . . .	217
13.6.1	Approche . . . . .	217

---

13.6.2	Problématique . . . . .	218
13.6.3	Test concolique <i>grey-box</i> . . . . .	218
13.6.4	Architecture de Eclipser . . . . .	219
13.6.5	Implémentation . . . . .	220
13.6.6	Résultats . . . . .	220
13.6.7	Conclusion . . . . .	221
<b>Chapitre 14</b>	<b>Évaluation des outils</b>	<b>223</b>
14.1	Bombes logiques : Évaluation & Analyse des résultats . . . . .	223
14.1.1	Résultat de UT_1 . . . . .	224
14.1.2	Résultats de UT_2 . . . . .	224
14.1.3	Résultats de UT_3 . . . . .	226
14.1.4	Résultats du UT_4 . . . . .	226
14.2	LAVA-M : Évaluation & Analyse des résultats . . . . .	226
14.3	Synthèse & Conclusion des benchmarks . . . . .	227
<b>Part VI</b>	<b>Slicing</b>	<b>229</b>
<b>Chapitre 15</b>	<b>État de l'art : Slicing</b>	<b>231</b>
15.1	Introduction . . . . .	231
15.2	Algorithmes . . . . .	232
15.3	Slicing statique . . . . .	232
15.3.1	Slicing à base d'interprétation abstraite . . . . .	233
15.3.2	Slicing à base de dépendance de programme . . . . .	233
15.4	Slicing dynamique . . . . .	238
15.5	Autres approches de slicing . . . . .	238

*Table des matières*

---

15.6 Applications du slicing . . . . .	239
15.6.1 Fuzzing . . . . .	240
15.6.2 Exécution symbolique . . . . .	240
<b>Chapitre 16 Analyse détaillée des outils</b>	<b>243</b>
16.1 Slicing #1 : CodeSurfer . . . . .	245
16.1.1 Fonctionnalités . . . . .	245
16.1.2 Algorithme de calcul du SDG . . . . .	245
16.1.3 Slicing au niveau binaire . . . . .	246
16.1.4 Utilisation . . . . .	247
16.2 Slicing #2 : Symbiotic . . . . .	248
16.2.1 Approche . . . . .	248
16.2.2 Slicer dg . . . . .	249
16.3 Synthèse & Conclusion . . . . .	252
<b>Part VII Conclusion</b>	<b>253</b>
<b>Chapitre 17 Propositions de combinaisons</b>	<b>255</b>
17.1 Interfaçage avec Klocwork . . . . .	256
17.2 Proposition #1 . . . . .	256
17.3 Proposition #2 . . . . .	257
17.4 Proposition #3 . . . . .	258
17.5 Fonctionnalités complémentaires . . . . .	259
<b>Chapitre 18 Conclusion Générale</b>	<b>261</b>
<b>Acronymes</b>	<b>263</b>

---

Glossaire	267
Bibliographie	271
Annexe	305
Annexe A Machines de Mealy	305
Annexe B Compétition Rode0day	307
Annexe C Sail : Instruction Set Architecture (ISA) ARMv8-A	309
Annexe D Différence approche CP et SMT	311
Annexe E Résolution des débordements de buffer par angr	313
Annexe F Différence analyse de teinte et slicing en avant	315
Annexe G Métrique du nombre d'exécution par seconde	317

*Table des matières*

---

# Première partie

## Introduction



# Chapitre 1

---

## Information sur le projet

---

### 1.1 Contacts

Côté Quarkslab, plusieurs ingénieurs sont intervenus sur différentes catégories d'analyses ou d'outils en fonction de leurs compétences et de leurs centres d'intérêt respectifs. Tous les ingénieurs ayant participé à la rédaction de ce rapport sont mentionnés dans le tableau 1.1.

Quarkslab	Fonction	Coordinées
Frédéric Raynal	Président	+33 (0)6 79 74 19 37 <a href="mailto:fraynal@quarkslab.com">fraynal@quarkslab.com</a>
Jean-Philippe Luyten	Chef de projet & Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:jpluyten@quarkslab.com">jpluyten@quarkslab.com</a>
Robin David	Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:rdaid@quarkslab.com">rdaid@quarkslab.com</a>
Paul Hernault	Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:phernault@quarkslab.com">phernault@quarkslab.com</a>
Christian Heitman	Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:cheitman@quarkslab.com">cheitman@quarkslab.com</a>
Boyan Milanov	Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:bmilanov@quarkslab.com">bmilanov@quarkslab.com</a>
Jonathan Salwan	Ingénieur R&D	+33 (0)1 58 30 81 51 <a href="mailto:jsalwan@quarkslab.com">jsalwan@quarkslab.com</a>

TABLE 1.1 – Contacts Quarkslab

*Chapitre 1. Information sur le projet*

---

## Chapitre 2

---

# Synthèse des résultats

---

## 2.1 Contexte

La DGA (Direction Générale de l'Armement, Maîtrise de l'Information (DGA-MI)) est en charge de l'évaluation de la sécurité de systèmes d'information et de systèmes d'arme. Dans ce cadre, DGA-MI assure une veille technologique sur les techniques et les outils d'investigation sur les systèmes en cours d'évaluation. Elle cherche à renforcer ses compétences et ses moyens d'évaluation de produits de sécurité, en particulier en matière de recherche de vulnérabilités au sein de logiciels embarqués.

L'objet de la présente étude est d'apporter des éléments d'informations complémentaires qui permettront à la Direction Générale de l'Armement (DGA) de mieux orienter les choix futurs d'outils mis en œuvre pour ses évaluations. Ce projet appelé PASTIS vise à la spécification et à l'implémentation d'un démonstrateur logiciel mettant en œuvre différents outils d'analyse et d'évaluation logicielle. Le projet se décompose en quatre postes qui visent à établir – dans l'ordre – un état de l'art des techniques et des outils d'analyse, à spécifier et à implémenter un démonstrateur, à transférer les connaissances à la DGA et enfin à ajouter une méthode de slicing au démonstrateur (poste optionnel).

L'innovation de ce projet réside dans l'association rationnelle et automatisée de plusieurs méthodes existantes pour obtenir un gain significatif par rapport aux méthodes actuelles dans la complétude de vérification de bugs et de vulnérabilités avérées, et ce, sans augmenter le temps mis pour les diagnostiquer. Plus précisément, l'idée est de combiner, à partir d'un code source à analyser présentant un certain nombre d'alertes dont le statut n'est pas vérifié, différentes techniques d'analyse pour les vérifier ou les invalider. Les combinaisons considérées sont l'analyse statique avec l'exécution concolique, l'exécution concolique avec le fuzzing, l'analyse statique avec l'exécution concolique et le fuzzing ; et enfin l'analyse statique avec le slicing, l'exécution concolique et le fuzzing. Le présent document correspond au livrable du poste 1.

## 2.2 Plan du rapport

Cette étude s'articule autour des trois techniques d'analyse centrales (exécution concolique, fuzzing, slicing) et d'une base de tests destinée à faire ressortir les différences entre les techniques et les outils qui les mettent en oeuvre. Le choix de la base de tests et son élaboration sont dûment documentés dans la Partie II. Cela comprend une comparaison des différentes bases à disposition, le choix effectué, les critères d'évaluations qu'elle teste ainsi que les conditions d'exécution des tests (cf. 5.5). Les parties III, IV et VI présentent respectivement les techniques et les outils d'exécution concoliques, de fuzzing et de slicing.

Pour chacune de ces trois dernières parties, un état de l'art général présentant les différents aspects et les particularités de la technique sont présentés afin de dresser une vision globale *au-delà des outils*. Ensuite, parmi les différents outils analysés succinctement, un choix argumenté est effectué pour une phase d'étude plus approfondie. Chacun des outils sélectionnés est alors documenté et décortiqué pour en expliquer le fonctionnement à partir des ressources disponibles à son sujet. Si l'outil est disponible, il est testé sur la base de tests. Une synthèse mettant en relief les avantages et inconvénients de chaque outil est formulée en conclusion de ces parties.

La partie V présente de la même manière les différentes techniques de combinaisons. La difficulté de cette partie est d'unifier de manière cohérente la multitude de combinaisons de techniques parfois très hétérogènes. La comparaison des outils entre eux est donc beaucoup moins pertinente que pour les parties précédentes. Pour certains outils le choix de les catégoriser comme relevant d'une seule technique ou d'une combinaison a été effectué de manière subjective. Par exemple, il est difficile de qualifier de combinaison un outil effectuant une passe de slicing dans son moteur interne. Les fuzzers dits "whitebox" sont un autre exemple, ils ont été qualifiés dans cette étude de combinaison et non de fuzzers au sens strict.

Enfin la partie VII, qui conclut le rapport, présente les différentes propositions de combinaisons (chap. 17) qui paraissent les plus pertinentes au vu des benchmarks effectués précédemment et en vue d'une intégration dans le démonstrateur, elle s'achève par une conclusion générale sur l'étude 18.

## 2.3 Synthèse de l'étude

Les ressources et les travaux intéressants pour le projet PASTIS sont extraordinairement abondants. Inversement, le besoin et les exigences du démonstrateur sont très spécifiques. Il a donc été particulièrement difficile de cerner et d'identifier les travaux appropriés aux besoins du projet, sachant qu'aucun des outils ne fournit toutes les fonctionnalités désirées (*sans être commercial et coûteux*). En outre, pour une ressource donnée il a été épineux d'en évaluer le bien-fondé et l'usage qui pourrait être fait dans l'étude au seul vu des résultats annoncés dans une publication ou une plaquette commerciale.

D'autre part, l'évaluation a été beaucoup plus délicate qu'initialement envisagé. En effet, il a été difficile d'élaborer un test qui soit adapté aux deux usages attendus (*exécution symbolique et fuzzing*). Par ailleurs, les spécificités de chaque outil nécessitaient pour la plupart d'utiliser leur compilateur. A titre d'exemple, certains fuzzers activent les optimisations maximums, qui, appliquées sans précaution, peuvent entraver le test et fausser les résultats.

Enfin, il s'est révélé particulièrement difficile de prendre en main et de tirer le plein potentiel de chacun des outils en un temps très contraint. En effet, s'approprier et comprendre le fonctionnement d'un outil requiert souvent beaucoup de temps pour extraire des résultats concluants.

**DSE** Parmi les ressources de DSE étudiées, seuls `angr`, `Manticore`, `Triton` et `KLEE` possèdent un outil idoine au besoin du projet. `Mayhem` a été ignoré, car il n'est pas public. `angr` et `KLEE` se sont révélés très polyvalents et semblent agréger au fur et à mesure les dernières techniques à l'état de l'art. Les communautés qu'ils agglomèrent commencent à porter leurs fruits. `Triton`, de par la parfaite connaissance que Quarkslab en a, est résolument l'outil ayant fourni les meilleurs résultats et le plus adapté pour une combinaison avec une autre technique du fait de son fonctionnement sous forme de bibliothèque.

**Fuzzing** Parmi les fuzzers, malgré une littérature très importante, peu d'outils sont disponibles et utilisables. De surcroît, aucun d'entre eux ne satisfait directement les exigences du projet. Les points limitants restent le support de l'architecture ARM et surtout le support du fuzzing réseau (au niveau des couches basses). Toutefois `Honggfuzz` a fourni de très bons résultats et, malgré le doute qui plane concernant son support de ARM et du réseau, il reste le candidat le plus sérieux moyennant quelques modifications et améliorations.

**Combinaisons** Les combinaisons d'outils sont très variées et représentent l'essentiel des publications récentes. Fort du constat des faiblesses respectives des différentes approches, la combinaison d'analyse tend à devenir la norme, et les résultats obtenus sont significatifs. `Qsym` et `Angora` valident l'idée qu'une combinaison soigneusement conçue d'outils collaborants en profondeur produit des résultats qui dépassent les outils pris indépendamment. Au-delà des considérations liées au langage (e.g `Angora` en Rust), utiliser une combinaison existante comme base pour `PASTIS` est très judicieux. Le dernier venu, `Eclipser`, propose une approche très innovante, digne d'intérêt, et a fourni de très bons résultats sur la suite de test.

**Slicing** Le slicing a été relativement difficile à étudier, car c'est un but et non pas un moyen. Il peut être statique, dynamique, s'articuler autour d'une analyse de teinte, une interprétation abstraite ou tout autre critère issu de différentes métriques. De surcroît, le slicing est souvent utilisé sans le dire dans des passes internes d'outils pour, par

exemple, guider la couverture. Parmi les ressources étudiées, seul dg [323] a pu être testé. Au vu des résultats obtenus, l'approche la plus rationnelle pour l'utilisation du slicing serait d'implémenter directement une analyse *ad hoc* inspirée de l'état de l'art, au sein du démonstrateur.

Il est à noter que les résultats de certains outils considérés comme leaders du domaine (comme AFL) se sont révélés bien en deçà d'autres outils moins connus, mais plus récents. Cette tendance confirme qu'un outil ne recevant pas un support et un développement actif finit par être dépassé par d'autres outils qui implémentent les dernières techniques à l'état de l'art – qui est en constante évolution. Par ailleurs, peu d'outils se sont révélés utilisables facilement “*out-of-the-box*” et encore moins se prêtent à une combinaison avec d'autres outils sans une refonte partielle ou totale.

## 2.4 Synthèse des propositions de combinaisons

À l'issue de l'étude des dix outils sélectionnés et des dernières techniques de l'état de l'art, trois propositions de combinaison ont été formulées. À la lumière des résultats obtenus, ce sont assurément les plus appropriées pour remplir les tâches dévolues au démonstrateur. La première combinaison (cf.17.2) allie Honggfuzz [335] et Triton [304] dans une approche de DSE sélectif [333]. Triton, embarqué directement dans le fuzzer ou agissant comme un composant distinct, serait en charge de résoudre les chemins “difficiles” rencontrés par Honggfuzz. Le choix de Honggfuzz comme fuzzer résulte du fait qu'il est le seul à satisfaire les contraintes imposées par le projet. La deuxième proposition (cf.17.3), combine Honggfuzz et KLEE dans une approche asynchrone où les deux outils enrichissent respectivement le répertoire de travail (*workspace*) de l'autre avec les entrées, les chemins et autres connaissances acquises durant l'exécution. Enfin, la dernière combinaison (cf.17.4) s'articule autour de l'architecture de Qsym [380] en changeant certains composants pour le rendre compatible avec les besoins du projet, et en particulier le support de ARM. Ainsi, AFL serait remplacé par Honggfuzz, l'instrumentation remplacée par QBDI [181] et la génération des expressions symboliques par Triton. Cette combinaison est certainement la plus ambitieuse, mais aussi la plus prometteuse.

À chacune de ces trois combinaisons est ajoutée une proposition visant à intégrer une technique ou un outil de slicing. Pour chaque combinaison une méthode de slicing pourra être greffée en cas d'activation du poste 4.

# Chapitre 3

---

## Introduction

---

Le premier papier mentionnant le terme fuzzing par Barton P.Miller [251] effectuait déjà le constat de la nécessité de techniques automatisées d’analyse de programme dans les années 90. Le constat était alors :

1. Les programmeurs ne vérifient ni les bornes de tableaux ni les codes d’erreurs ;
2. les macros rendent le code difficile à la lire et debugger ;
3. le C est hautement risqué (non sûr).

Ce constat s’effectuait sur le C alors prépondérant. L’avènement de langages plus fortement typés et fournissant de meilleures propriétés de sécurité lors de la compilation limite ce genre de problèmes. Cependant, ce constat est toujours autant d’actualité et la quantité s’étant décuplées il est plus que jamais nécessaire d’avoir recours à des analyses automatisées de programme – tâche autrement manuellement impossible.

### 3.1 Analyses automatisées de programmes

#### 3.1.1 Aperçu des techniques d’analyse existantes

Cet état de l’art étudie les différentes techniques de générations de tests appliquées à la découverte de bugs. Parmi celles-ci, il est possible de mentionner le test aléatoire *random testing* [8], l’exécution symbolique [19], l’exécution concolique [284], le test par recherche *search-based* [165] ou encore le test par modèle [157]. Ce dernier est populaire, mais souvent irréalisable, car peu de logiciels sont développés avec des modèles et une spécification. Le *random-testing* amène souvent de nombreux cas de test redondants avec une faible couverture. Cette technique est une approche naïve et a maintenant évolué en fuzzing où la génération des cas de tests n’est plus aléatoire, mais conditionnée par différents algorithmes.

Ces différentes techniques se caractérisent par différents paramètres comme la direction de l'analyse (avant/arrière), le niveau d'approximation, la modularité ou encore la correction. Le tableau 3.1 dresse un aperçu des caractéristiques pour l'interprétation abstraite (noté AI), le calcul de plus faible précondition Weakest-Precondition Calculus (WP), l'exécution concolique aussi appelée exécution symbolique dynamique (noté DSE), le fuzzing, l'exécution symbolique statique (noté SSE) et le slicing qui diffère énormément selon l'approche.

Critère	IA	WP	DSE	Fuzzing	SSE	Slicing	
						statique	dynamique
<b>Nature</b>	statique	statique	dynamique	dynamique	statique	statique	dynamique
<b>Flot</b>	donnée	donnée	les deux	contrôle	donnée	donnée	contrôle
<b>Direction</b>	avant	arrière	avant <sup>1</sup>	avant	avant	arrière	arrière
<b>Type</b>	sémantique	sémantique	sémantique	/	sémantique	syntaxique	syntaxique
<b>Modularité</b>	programme	fonction	programme	les deux	fonction	les deux	programme
<b>Approximation</b>	sur-app.	sur-app.	sous-app.	sous-app.	sur-app.	sous-app.	sous-app.
<b>Correction</b>	✓	✓	✓	✓	✓	✗	✗
<b>Complétude</b>	✓	✓	✗	✗	✓	✗	✗

TABLE 3.1 – Caractéristiques des types d'analyses

Dans le tableau la *Nature* définit le type d'analyse entre statique et dynamique, le *Flot* définit le flot sur lequel l'analyse effectue son calcul. La *Direction* indique le sens de l'analyse. En arrière, l'analyse part d'un point d'intérêt (*ou goal*) et remonte en arrière sur le flot. On dit de l'approche qu'elle est *goal-oriented*. Le *Type* définit le type de données manipulées. Une analyse sémantique considère le comportement ou effet de bord d'une instruction tandis qu'une analyse syntaxique considère seulement sa représentation. La *Modularité* définit à quel niveau s'effectue l'analyse. La plupart des analyses considèrent le programme en entier, mais des analyses plus coûteuses vont par exemple effectuer l'analyse fonction par fonction.

L'*Approximation* définit la manière dont l'analyse considère tous les états possibles du programme. Soit elle est sur approximée – considère tous les états possibles *au détriment de la précision* – soit elle ne considère qu'un sous-ensemble et dans ce cas elle est sous-approximée. L'avantage en termes de sûreté d'une analyse sur approximée est que si l'analyse prouve l'absence de bugs, il est certain qu'il n'existe pas d'états mémoire symptomatiques d'un bug. Dans un contexte de recherche de vulnérabilité, l'analyste cherche plutôt un seul cas de test exerçant le bug, les analyses sous-approximées étant plus précise cela limite donc grandement les faux positifs. En résumé, une sur approximation possède potentiellement de nombreux faux positifs, mais aucun faux négatif alors qu'une sous-approximation minimise les faux positifs au détriment des faux négatifs potentiellement nombreux.

Enfin, *Correction* et *Complétude* viennent étayer les assurances fournies concernant les résultats de l'analyse. Dans un contexte de génération de cas de test, une analyse est dite

---

1. quelques-uns en arrière

correcte si les entrées générées permettent de couvrir le chemin emprunté. Elle est dite complète si un cas de test est généré pour tous les chemins qui peuvent être empruntés. Une définition plus précise et spécifique à chaque type d'analyses est fournie dans la partie idoine. En effet les approximations effectuées par les différentes analyses ont un impact sur la correction et la complétude.

Pour conclure, ce tableau présente les caractéristiques de chaque type d'analyses dans l'usage qui en est couramment fait. En effet il existe une myriade de variantes pour chaque analyse. On peut trouver des WP interprocéduraux [67] ou encore une exécution symbolique avec postconditions [376] (*ce qui le rapproche du WP*). De même le DSE est théoriquement correct et complet, mais en pratique des approximations sont toujours effectuées et tous les chemins ne peuvent pas être couverts en pratique. Il en va de même avec combinaisons d'analyses qui brouillent encore plus les caractéristiques d'un outil donné pour lesquels il est difficile de définir les critères énoncés ci-dessus.

### 3.1.2 Techniques d'analyse pour PASTIS

Parmi les différentes approches d'analyses présentées dans la section précédente, l'exécution symbolique et le fuzzing se présentent comme des solutions de choix (*bien que n'étant pas immune de limitations*). En effet ces deux approches sont résolument orientées sécurité alors que d'autres approches comme l'interprétation abstraite ou le WP sont plus orientés sûreté – ce dernier requiert par exemple des annotations manuelles.

Pour les besoins du projet PASTIS seules les techniques d'analyse concolique, de fuzzing et de slicing sont considérées – l'analyse statique étant effectuée par Klocwork. Ces trois techniques jouissent d'un dynamisme sans précédent dans le domaine de la sécurité pour lequel elles apportent de très bons résultats. Le nombre de bugs trouvés via ces techniques est relativement élevé. Ce domaine a notamment tiré profit du challenge Cyber Grand Challenge (CGC)<sup>2</sup> [242] organisé par la Defense Advanced Research Projects Agency (DARPA)<sup>3</sup>, de 2013 à 2016 duquel beaucoup d'innovations et d'outils ont été conçus (e.g angr [320], Manticore [33]) grâce aux financements.

## 3.2 Questions de recherche

Le projet PASTIS soulève de nombreux problèmes de recherche encore non résolus à ce jour. Les problématiques de recherche liées à l'exécution concolique, au fuzzing, au slicing et à la combinaison d'analyses sont dûment énoncées et résumées dans chacune des parties. Les tableaux de référence sont 6.5, 9.3 et 12.3 pour chacun des types d'analyse.

La plupart des problématiques sont liées au passage à l'échelle des analyses qui par la nature non finie des programmes ne peuvent être complètes. Aussi bien le DSE que

---

2. <https://www.darpa.mil/program/cyber-grand-challenge>

3. <https://www.darpa.mil/>

le fuzzing ne seront pas en mesure de couvrir l'entièreté du code dans le temps imparti pour un programme de taille normale. L'autre aspect sur lequel se posent les problèmes de recherche est la précision des résultats. Sans être sur approximé le but est quand même de minimiser les faux positifs et les faux négatifs dont l'un se fait souvent au détriment de l'autre.

### 3.3 Définitions de vulnérabilité

La RFC 2828 [318] fournit la définition de vulnérabilité suivante :

**Définition 1.** *Toute faille ou défaut dans la conception, l'implémentation ou le fonctionnement d'un système qui pourrait être exploité pour en violer la politique de sécurité.*

Cette définition est relativement vague, mais introduit la notion de politique de sécurité nécessaire pour différencier un bug ou défaut d'une vulnérabilité. Dans le contexte d'analyse de programme la politique de sécurité standard est que tout défaut menant à l'exécution d'un chemin non prévu par le développeur est une violation de la politique de sécurité et donc une vulnérabilité. Cette définition implique donc nécessairement un contrôle par l'attaquant du pointeur d'instruction pour qu'un bug soit qualifié de vulnérabilité. Par abus de langage, dans ce rapport, un bug peut-être appelé vulnérabilité et inversement.

### 3.4 Sélection des documents

L'exécution symbolique, le fuzzing et les différentes combinaisons touchent un très grand nombre de publications qu'il fut nécessaire de filtrer. Pour les publications académiques, le principal choix fut le rang de la conférence dans laquelle le papier a été publié. Le système de revue par les pairs *peer-reviewing* est un gage d'une certaine qualité. Ainsi, dans le domaine de la sécurité les conférences *Security & Privacy* (S&P), *Computer and Communications Security* (CCS), *Network and Distributed System Security Symposium* (NDSS) ou encore *USENIX Security Symposium* ont été des conférences de choix. Dans le domaine industriel, Quarkslab s'est appuyé sur ses connaissances et son expérience dans le domaine afin d'étudier les outils les plus pertinents. D'autre part, cet état de l'art se focalise prioritairement sur les outils utilisés à des fins de sécurité et de surcroît à des fins de recherche de vulnérabilités. Les outils d'exécution symbolique ou de fuzzing utilisés uniquement à des fins de test logiciel pour de la sûreté ont été écartés.

Les sources documentaires considérées ont été aussi exhaustives que possible. La différence fut le niveau de lecture appliqué à chacune. Dans les différents états de l'art, les ressources ont pour la plupart été parcourues afin d'en extraire les notions clés. À l'inverse pour les outils sélectionnés un examen du principal papier a été effectué pour en saisir le fonctionnement avec le plus de précision possible.

## 3.5 Méthodologie d'analyse

Pour chacun des types d'analyses, un premier travail prospectif a d'abord été effectué pour cerner les différents aspects et idiosyncrasies inhérentes la technique d'analyse. À partir de cela les différentes ressources faisant autorité et satisfaisant à priori la plupart des exigences ont été sélectionnées afin d'être étudiées.

Pour chaque ressource sélectionnée, un temps d'étude total de 5 jours a été alloué (*conformément à la proposition technique*). Ce temps englobe une première lecture, une lecture approfondie, la prise en main d'un éventuel outil, le lancement des benchmarks et la rédaction détaillée du fonctionnement et les résultats obtenus.

### Gestion du temps

Le temps alloué à chaque ressource étant extrêmement contraint il fut extrêmement difficile de se conformer au temps imparti. Plusieurs facteurs hasardeux sont parfois venus consumer plus de temps que prévu, par exemple, la difficulté de prise en main de l'outil, la compréhension de bugs ou encore l'adaptation de la suite de tests.

Après l'étude de chaque ressource d'une technique d'analyse, un travail de consolidation a été effectué pour confronter les résultats entre les outils afin d'en extraire les candidats les plus prometteurs.

### Implémentation de “driver”

Si nécessaire un “driver” est développé pour permettre d'exécuter l'outil sur la suite de test. En effet pour cibler un composant ou une fonction particulière dans un programme il est souvent nécessaire d'implémenter un “driver” (programme mère) faisant l'interface entre le fuzzer et la fonctionnalité du programme ciblé. C'est particulièrement le cas lorsque le programme à analyser une librairie dynamique. Ce processus étant très largement manuel, ainsi, la suite de test choisie (décrise au chapitre 5) a été élaborée pour limiter au maximum d'avoir recours à un driver.



## **Deuxième partie**

### **Base de tests**



# Chapitre 4

---

## Etat de l'art : Base de tests

---

### 4.1 Introduction

Établir une base de tests et évaluer les outils sur celle-ci est une tâche très fastidieuse. En effet, cela pose de nombreux problèmes. Il faut que la base de test soit représentative de vrais programmes tout en restant testable en un temps raisonnable. Elle doit être objective et ne pas favoriser un outil plutôt qu'un autre. Mais surtout, pour être une base de référence, elle doit être adoptée et utilisée dans un grand nombre d'études. Or il n'existe pas de base de tests universellement acceptée et utilisée par tous. De récents travaux [211] ont remis en cause la méthodologie de *benchmarking* des outils de fuzzing et dégagent dix critères qui devraient être satisfaits pour une étude significative et objective. Ces critères, résumés dans un post de blog de Trail of Bits<sup>4</sup>, sont les suivants :

1. **Comparaison sur les outils de références**, comme AFL [383] libfuzzer [338] ou encore Honggfuzz [335] ;
2. **La sortie doit être facile à lire et comparer**. Cela implique d'avoir des valeurs de référence (*ground-truth*) et la possibilité de discerner un faux positif, un faux négatif ou les duplicita d'un même bug ;
3. **Prendre en compte les différences d'heuristique**. La base et l'évaluation doivent prendre en compte que certaines heuristiques utilisées dans les outils sont susceptibles de les favoriser ;
4. **Utiliser des jeux de donnés avec des bugs distinguables**, comme les binaires du CGC (cf. Section 4.2) ou la suite LAVA-M (cf. Section 4.3) ;
5. **Configuration initiale identique**. Les fuzzers et outils doivent, dans la mesure du possible, utiliser une configuration initiale identique ;
6. **Timeout d'au moins 24h**. Pour lisser l'aléa statistique lié à des campagnes de fuzzing courtes ;

---

4. <https://blog.trailofbits.com/2018/10/05/how-to-spot-good-fuzzing-research/>

7. **Distinction claire de l'unicité d'un bug.** L'unicité d'un crash ne fait pas consensus. Pour un même crash est-ce l'entrée utilisée, le chemin emprunté ou encore la pile d'appels qui différencient deux bugs d'un bug unique ?
8. **Choix uniforme de la graine,** tous les outils doivent utiliser le même corpus initial ;
9. **Répétition des tests (30 fois)** la méthodologie devrait répéter chaque test 30 fois et calculer la variance entre chaque “run”
10. **Préférer le nombre de bugs à la couverture.** Le critère du nombre de bug est plus proche du but d'une campagne, qui est de trouver des bugs. Une couverture aussi bonne soit-elle, peut ne pas déclencher les bugs ;

La base de tests et la méthodologie employée dans cette étude tentent de satisfaire au mieux ces différents critères, mais tiennent aussi compte des contraintes de temps alloué à l'étude de chaque ressource.

## 4.2 Etat de l'art des bases existantes

L'évaluation d'outils d'analyse automatisée se heurte depuis longtemps à la problématique de la base de tests qui doit être pertinente et représentative des programmes rencontrés en production. Dans le domaine du test logiciel orienté “sûreté” une suite sortie en 1994 largement utilisée fut celle de Siemens Corporate Research [182] avec notamment **tcas** un système anticollision aérien ou différents programmes d'ordonnancement ou de parsing. Cette suite ciblait le test des critères de couverture de contrôle et de données. Ultérieurement d'autres suites telles que Verisec [216], MediaBench<sup>5</sup> [227], Toyota ITC bench<sup>6</sup> [317] ou encore SPEC<sup>7</sup> ont vu le jour. Cette dernière, payante, a l'ambition de s'établir comme une base de référence commune. En effet, une suite n'est pertinente que si elle est adoptée par les outils d'analyse. À noter que la plupart d'entre elles ont été conçues pour tester des analyseurs statiques et non pas des techniques de fuzzing ou d'exécution concolique.

Des organismes tels que le NIST supervisent des suites de tests via notamment le projet **Software Assurance Reference Dataset Project (SARD)**<sup>8</sup>. Avec notamment la base de tests SAMATE [38] et Juliet [42] dont l'un des principaux contributeurs est le **Center for Assured Software (CAS)** de la **National Security Agency (NSA)**. Ce projet héberge aussi une suite proposée par Klocwork utilisée en interne pour leurs tests de régression<sup>9</sup>. La naïveté des tests élimine cependant *de facto* son utilisation comme base de test. De plus celle-ci semble uniquement dédiée à des analyseurs statiques. L'exemple 1 illustre la trop grande simplicité des tests. On peut y voir comment un bug est inconditionnellement activé.

---

5. <http://mathstat.slu.edu/~fritts/mediabench/>  
6. <https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark>  
7. <https://www.spec.org/benchmarks.html>  
8. <https://samate.nist.gov/SARD/testsuite.php>  
9. <https://samate.nist.gov/SARD/view.php?tsID=106>

---

**Listing 1** Exemple `uninit_014.c` de la suite Kocwork du NIST ([SARD](#))

```

1 int arr[5];
2 int main(){
3     int i;
4     arr[i++]=1;
5 }
```

---

Ces bases de tests génériques n'ont pas nécessairement vocation à être utilisées pour tester des outils de DSE et de fuzzing. Des bases plus spécifiques aux tests de sécurité et à la recherche de vulnérabilités par fuzzing ou DSE ont été publiées ces dernières années. Voici différentes bases proposées comme références pour la recherche de vulnérabilités.

**Test du challenge CGC** Le [CGC](#) est une compétition qui fut organisée par la [DARPA](#)<sup>10</sup> dont le but était d'élaborer un [Cyber Reasoning System \(CRS\)](#) capable de trouver des vulnérabilités, de les corriger et de s'en servir pour attaquer d'autres concurrents, et ce de manière complètement automatisée. Cette compétition a énormément dynamisé la recherche et les avancées dans le domaine de la recherche de vulnérabilité. De nombreux outils issus de cette compétition ont vu le jour et l'ensemble des programmes de test utilisés pendant la compétition sont disponibles. Les programmes de test ont la particularité de fonctionner sur DECREE, un système Linux minimaliste sans signaux, mémoire partagée, threads, runtime de libc et uniquement 7 appels système. Le corpus<sup>11</sup> contient 245 programmes<sup>12</sup> "réalistes" dans lesquels ont été ajoutés différents bugs pour lesquels il est théoriquement possible de générer une [Proof Of Vulnerability \(POV\)](#). Tous les tests ont été portés par Trail of Bits [154] pour fonctionner sur Linux Windows et macOS et sont disponibles sur leur Github<sup>13</sup>. Cette suite se positionne comme successeur des suites SAMATE ou Juliet du NIST.

**Logic Bombs**<sup>14</sup> Cette suite a été créée spécifiquement pour tester les outils d'exécution symbolique [370] et en particulier Triton, angr et KLEE pour lesquels des facilités de compilation sont fournies. Cette suite introduit l'idée de bombe logique pour lequel un programme test renvoie soit `BOMB_ENDING` soit `NORMAL_ENDING` selon que la "bombe" a été exécutée ou non à l'intérieur du test. Cette suite contient 53 fichiers C organisés dans les catégories suivantes :

- déclaration de variables symboliques : déclaration de nouveaux symboles sur des appels de la libc ;

---

10. <https://www.darpa.mil/program/cyber-grand-challenge>

11. <http://www.lungetech.com/cgc-corpus/about/>

12. [https://docs.google.com/spreadsheets/d/1Z2pinCk0qe1exzpvFgwSG2wH3Z09LP9VJk0bm\\_5jPe4/edit#gid=520069001](https://docs.google.com/spreadsheets/d/1Z2pinCk0qe1exzpvFgwSG2wH3Z09LP9VJk0bm_5jPe4/edit#gid=520069001)

13. <https://github.com/trailofbits/cb-multios>

14. [https://github.com/hxuhack/logic\\_bombs](https://github.com/hxuhack/logic_bombs)

- valeurs symboliques contextuelles : valeur contextuelle (e.g l'ouverture d'un fichier ne renvoie un *handle valide que si celui-ci existe*) ;
- propagation indirecte : propagation des entrées du programme (e.g : écriture puis lecture dans un même fichier) ;
- mémoire symbolique : lecture à des index contrôlés par l'utilisateur ;
- programmes concurrents : gestion de la concurrence par le moteur ;
- nombres flottants : gestion des nombres flottants et de certaines fonctions de la libc ;
- sauts symboliques : sauts à des index symboliques (e.g : un switch) ;
- débordement d'entiers : débordement d'entiers via des additions ou multiplications ;
- débordement de tampons : débordement de tampons ;
- fonctions externes : support de fonctions de la libc ;
- fonctions cryptographiques : support des fonctions cryptographiques (sha1 et AES) ;
- boucles : gestion de la couverture des boucles.

Les tests prennent leurs entrées soit en ligne de commandes soit depuis `stdin`. Chaque test vise à vérifier avec du code minimaliste le comportement des moteurs de DSE vis-à-vis de diverses constructions d'un programme. Le bien-fondé des catégories et de chacun des tests est discuté en section 5.3.

**program-verification-samples**<sup>15</sup> Suite de test privée inspirée de “logic bombs” développée par Jonathan Salwan à des fins de test internes. Elle se compose de 30 fichiers organisés selon les catégories suivantes :

- tableaux (*9 tests*) ;
- sauts dynamiques (*1 test*) ;
- boucles (*5 tests*) ;
- débordements
  - tampons (*2 tests*)
  - entiers (*2 tests*) ;
- chemins (*6 tests*) ;
- propagation (*3 tests*) ;
- chaîne de caractères (strings) (*2 tests*) .

Cette suite permet notamment la récupération automatique des résultats des tests.

**fuzzer-test-suite**<sup>16</sup> est une suite utilisée par Google pour tester `libfuzzer` et `AFL`. L'avantage est qu'elle se base sur de vrais programmes tels que `openssl`, `sqlite`, `libxml` ou encore `boringssl` et plus particulièrement sur de vraies CVE qui ont été présentes dans ceux-ci. Ils fournissent par ailleurs un outil de test A/B permettant de comparer deux configurations d'un même fuzzer<sup>17</sup>. En revanche, le principal inconvénient de cette suite est qu'elle est initialement dédiée au fuzzing, et cible en particulier `libfuzzer` et `AFL`.

---

15. <https://github.com/JonathanSalwan/program-verification-samples>

16. <https://github.com/google/fuzzer-test-suite>

17. <https://github.com/google/fuzzer-test-suite/tree/master/engine-comparison>

## 4.2. Etat de l'art des bases existantes

**BugZoo**<sup>18</sup> élaborée par Squares [342] elle vise notamment la reproductibilité dans le temps des tests grâce à des conteneurs docker. La suite de test comprend les programmes **gmp**, **gzip**, **libtiff**, **lighttpd**, **php**, **python** et **wireshark**, mais l'approche se veux collaborative grâce à la flexibilité de docker. Publiée en Juin 2018 cette suite bien que prometteuse n'a pas encore été adoptée par la communauté.

**Suite Hemiptera**<sup>19</sup> Cette suite accessible sur demande a été créée par John Galea et Sean Heelan en se basant sur le constat que la recherche académique s'appuie trop souvent sur les coreutils au lieu d'utiliser de vrais programmes. Cette suite s'appuie sur différents projets open source dans lesquels des CVEs ont été trouvées. Les binaires sont **jasper**, **file**, **zlib**, **libjpeg-turbo**, **libgd**, **tcpdump**, **libxml2**, **libtiff**, **w3m**, **flac**, **libtasn** et **audiofile**. Tous les bugs ont été documentés avec le numéro de commit, la localisation dans le binaire et surtout les entrées permettant de les déclencher. La base de tests représente pour chaque bug 216 cas de tests, dont 144 pour lesquels des entrées permettent de confirmer le bug. La suite est documentée dans un fichier excel accessible sur invitation et le Github fournit quelques facilitées pour créer la suite qui n'est pas complètement clé en main.



		Test Cases	216	
		TOTAL Num Of Confirmed Bugs with test cases:	144	
Project	Potential Crashes	Confirmed Crashes	Status	Min-Set
Jasper	22	19	Finished	<b>Mini-Set:</b> - Commit b702259 (ImgInfo) (18 Bugs) - Commit ed355a6 (Jasper) (2 Bugs)
file	7	7	Finished	<b>Optional</b> - Commit: b293f98 <b>Min-Set:</b> - Commit: b6e8437 (6 Bugs) - Commit: 4a51454 (2 Bugs) - Commit: 7445748 (1 Bugs)
zlib	3	3	Finished	<b>Optional</b> <b>Min-Set:</b> - Commit: 14763ac (with minigzip) (1 Bug) - Commit: 7c2a874 (with minigzip) (2 Bug)
libjpeg-turbo	4	2	Finished	<b>Optional</b> <b>Min-Set:</b> - Commit: 3091354 (cjepg) (2 Bug)
libgd	21	0	Ongoing	<b>Optional</b> <b>Min-Set:</b> - Commit: dae3ee9
tcpdump	14	11	Finished	<b>Min-Set:</b> - Commit: 8322d3a (1 bug) - Commit: a9e4211 (9 Bugs)
libxml2	1	0	Ongoing	
Coreutils		0	Ongoing	

FIGURE 4.1 – Feuille excel des tests de Hemiptera

18. <https://github.com/squaresLab/BugZoo>

19. <https://github.com/johnfxgalea/Hemiptera-Workshop>

**LinuxFlaw**<sup>20</sup> Cette suite répertorie une impressionnante liste d'identifiants de CVE et d'identifiants de Exploit DB (EDB) pour lesquels le concepteur a récupéré les différents Proof-of-Concept (PoC) permettant de déclencher chacun des bugs. Cette suite se concentre exclusivement sur les programmes Linux et fournit des tests pour **275 CVE** et **20 EDB**. Les tests avec les fichiers d'entrées sont fournis au niveau source, mais aussi en binaire via différentes machines virtuelles correspondant aux différents systèmes vulnérables Ubuntu, Fedora CentOS etc. Cette initiative est de loin la plus exhaustive et la plus complète pour une base de tests représentative de vulnérabilités réelles.

## 4.3 Génération automatique de bugs

Plutôt que de fournir une base de tests contenant des bugs, des approches récentes proposent des méthodes pour insérer des bugs automatiquement dans n'importe quel programme. Cette approche présente de nombreux avantages. Tout d'abord elle libère de la nécessité d'utiliser des CVE ou de créer des tests à la main qui seront forcément moins représentatifs qu'un vrai programme. Ensuite elle permet, en fonction de méthode d'ajout, de contrôler la quantité, la profondeur et la difficulté des bugs à trouver. Les principaux outils effectuant cette fonction sont Bug-Injector [203], EvilCoder [281] LAVA [109] qui fut le précurseur dans le domaine ou encore sont successeur sobrement appelé Apocalypse [297]. Les différents critères ci-dessous ont été proposés pour évaluer la pertinence d'un générateur automatique de bugs [297] :

- **Réaliste** : Le bug injecté doit être aussi réaliste que possible.
- **Profondeur** : Le bug doit être suffisamment profond. Autrement dit, il doit se trouver suffisamment intriqué dans les différentes conditions de branchement.
- **Décorrélé** : Chaque bug doit être indépendant d'un autre. La découverte d'un bug ne doit pas favoriser la découverte des autres.
- **Reproductible** : Le bug doit être atteignable et reproductible avec l'entrée l'ayant déclenché.
- **Rare** : Le bug ne doit être déclenché que par une faible proportion de toutes les entrées possibles du programme.

LAVA fut le premier en 2016 à proposer une technique automatisée d'ajout de vulnérabilités dans des programmes satisfaisant toutes les contraintes susmentionnées. Son principal atout est la facilité d'identifier le bug trouvé lors d'un crash, ainsi que les deux suites de test LAVA-1 et LAVA-M<sup>21</sup> qui, fournies clé en main, ont incontestablement contribué à sa large adoption. LAVA-1 fournit plusieurs versions du même binaire avec une seule vulnérabilité dans chaque programme, tandis que LAVA-M fournit 4 programmes (`base64`, `md5sum`, `uniq`, `who`) contenant chacun plusieurs bugs. LAVA se base sur de la propagation de teinte dynamique pour lier des octets d'entrée à des emplacements donnés du

---

20. <https://github.com/mudongliang/LinuxFlaw>

21. <http://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html>

programme. Ces octets doivent satisfaire la propriété dite de **Dead, Uncomplicated and Available (DUA)** c'est-à-dire qu'ils ne doivent pas déterminer le flot de contrôle et doivent être peu modifiés par le programme. Le mécanisme de teinte dynamique fonctionne avec PANDA [110], lui-même basé sur QEMU [26]. L'inconvénient de LAVA est qu'il ajoute principalement des dépassements de tampon et que pour satisfaire la propriété de rareté – déclenchement réduit à une fraction des entrées possible – il recourt à l'utilisation de patterns dits de *knob and trigger* qui s'appuient sur des *magic values*. Une valeur magique est généralement un entier 32 bits ayant une valeur particulière. Ce type de motif est très efficace contre les fuzzers actuels (cf. section 10.4.2).

EvilCoder<sup>22</sup> publié quelques mois après LAVA s'appuie lui aussi sur de la propagation de teinte, mais cette fois-ci calculée statiquement sur le source. Le but est aussi de trouver les connexions entre les entrées contrôlées de l'utilisateur et certaines positions du programme où des bugs peuvent être ajoutés. Cette propagation s'appuie sur Joern<sup>23</sup> un parseur C/C++ utilisant la technique dite de grammaire islandaise [254] et fournissant une représentation du programme sous forme de graphe de propriétés [372]. Ces graphes permettent l'analyse de teinte interprocédurale nécessaire à la propagation de la teinte statique. Cet outil a l'avantage de pouvoir ajouter des vulnérabilités de buffer-overflow, integer-overflow, fuite mémoire ou encore de format string. L'inconvénient de taille et qu'il n'assure pas l'atteignabilité et la reproductibilité des bugs insérés.

Apocalypse est une évolution de LAVA s'appuyant sur des techniques formelles d'exécution concolique de synthèse de programme par contraintes et d'énumération de modèles. Ces différentes analyses visent à ajouter des bugs plus réalistes, plus dépendants du contexte d'exécution du programme et ayant une chaîne de dépendance plus profonde pour être déclenchés. Cette technique utilise la notion de **Error Transition System (ETS)** qui représente une machine à états se rapprochant transition après transition vers l'état d'erreur. La transition finale est protégée par un prédictat synthétisé qui dépend du contexte et qui assure que seul un faible nombre de chemins d'exécution peuvent atteindre cet état. Apocalypse est très prometteur, mais il n'est à ce jour pas disponible publiquement.

Bug-Injector<sup>24</sup> est développé par GrammaTech et vise particulièrement à tester des analyseurs statiques. Il a été initialement développé à des fins internes. Il utilise un mécanisme de template décrivant une vulnérabilité ou une faiblesse **Common Weakness Enumeration (CWE)** auquel est greffé un certain nombre de conditions et une entrée déclenchant le bug. À partir de ces templates une instrumentation dynamique est effectuée pour ajouter des bugs à divers endroits pour les templates dont les préconditions sont satisfaites. L'instrumentation se base sur SEL<sup>25</sup> une chaîne de compilation en LISP fournissant des capacités d'instrumentation et de modification de logiciel.

---

22. <https://github.com/RUB-SysSec/EvilCoder>

23. <http://www.mlsec.org/joern/>

24. <https://go.grammotech.com/bug-injector/>

25. <https://grammotech.github.io/sel>

#### Attention

Les techniques d'insertion de bug à base d'exécution dynamique s'appuient *de facto* sur la capacité de couverture de l'exécution dynamique pour l'ajout de bugs. Aucun bug ne pourra être ajouté en dehors de ce qui est couvert. Ainsi, pour introduire des bugs de façon réaliste, l'analyse dynamique doit donc être en mesure de fournir une bonne couverture.

L'examen complet des différentes techniques d'injection de bugs et leur impact dans le contexte de ce projet ont été laissés hors périmètre de cet état de l'art. De plus les techniques de génération de bugs évoluent rapidement comme en attestent *Apocalypse* et *Bug-Injector* publiés respectivement le 21 novembre 2018 et le 1er février 2019.

## 4.4 Choix de la base de tests

La base de tests voulue par Quarkslab pour l'état de l'art devait satisfaire les deux besoins suivants :

- tester les comportements idiosyncratiques des outils avec des tests atomiques et contrôlés.
- tester le passage à l'échelle des outils sur des programmes de taille similaire à celui utilisé comme programme étalon.
- satisfaire au maximum les critères mentionnés par [211].

Pour les tests atomiques, Quarkslab s'est tourné vers les bombes logiques [370] et les tests de **program-verification-samples** auxquels ont été ajoutés différents tests permettant d'améliorer la compréhension du fonctionnement des outils.

Pour les tests de passage à l'échelle la suite LAVA-M de LAVA a été retenue comme candidat de choix. La principale raison est sa forte adoption dans le monde académique pour le benchmarking des outils de fuzzing. *Apocalypse* aurait probablement été plus pertinent, mais il vient juste d'être publié et aucune suite de test se basant dessus n'a encore été publiée. Une autre initiative favorisant l'utilisation de LAVA-M est la plate-forme *rode0day*<sup>26</sup> qui permet à différents outils de concourir en continu sur la base de programmes générés avec LAVA. L'annexe B donne un aperçu du fonctionnement de la plateforme.

Les autres suites de tests n'ont pas été retenues pour diverses raisons. La principale est que le temps imparti pour l'analyse de chaque outil ne permettait pas d'effectuer des benchmarks sur les 245 programmes du CGC ou les 275 CVEs de LinuxFlaw. L'accent a été mis sur la facilité d'identification des bugs de LAVA et les garanties de reproductibilité que celui-ci fournit. En termes de nombre de bugs, il fournit une base quantitative suffisamment discriminante pour les besoins de cet état de l'art.

---

26. <https://rode0day.mit.edu/>

# Chapitre 5

---

## Base de tests

---

### 5.1 Notations

Les programmes de la suite de test sont organisés en différentes catégories et sous-catégories ayant pour but de structurer de manière logique les critères testés. Pour cela, on note **UT** les différentes unités de test qui visent à tester les caractéristiques substantielles des outils. Parmi celles-ci, la modélisation mémoire, la gestion des entrées, l'aptitude à détecter de vulnérabilités et l'aptitude à passer à l'échelle sont des caractéristiques mises en relief par les unités de test.

Chaque **UT** est un compendium de critères d'évaluation notés **CE** qui visent à tester des propriétés plus spécifiques de l'unité de test courante. Certains **CE** sont plus spécifiques à certaines techniques (fuzzing, DSE), mais par souci d'homogénéité tous les **CE** sont communs à chaque outil.

Enfin chaque **CE** contient un certain nombre de cas de test notés **CId** qui identifient de manière unique chaque programme de test de la suite. Chaque programme vise à tester un aspect ou une construction différente de la **CE** courante.

### 5.2 Critères d'évaluation

Comme mentionné précédemment la difficulté posée par ce projet et cet état de l'art est d'établir des critères pertinents aussi bien pour le fuzzing que pour le DSE ou la combinaison des deux. En sus, la plupart des suites de test existantes sont conçues pour tester des analyseurs statiques ou uniquement le fuzzing ou le DSE (cf. 4). La structure globale de la base de tests est la suivante :

- **UT\_1** : Calcul de prédicat de chemin et modélisation mémoire
- **CE\_1 Tableaux** : Évalue les opérations sur des tableaux situés sur la pile, le tas, etc

- **CE\_2 Propagation** : Propagation des entrées utilisateur à travers des fichiers et variables
  - **CE\_3 : Lecture mémoire symbolique** : Tests de lecture en mémoire à des adresses symboliques
  - **CE\_4 : Écriture mémoire symbolique** : Tests d'écriture en mémoire à des adresses symboliques
  - **CE\_5 : Lecture/Écriture mémoire symbolique** : Test où l'utilisateur contrôle une primitive de lecture et écriture
- 
- UT\_2 : Symbolisation des entrées et modélisations des contraintes
    - **CE\_1 Fonctions externes** : Teste le support de certaines fonctions de la libc (et l'aptitude à l'outrepasser)
    - **CE\_2 Appels système** : Teste le support des appels système directement dans le programm, (`syscall` en x86 et `SVC` en ARM)
    - **CE\_3 : Instructions non déterministes** Teste l'aptitude à gérer des instructions ayant un comportement non déterministe (e.g : `rdtsc`, `mrs`)
    - **CE\_4 : Modélisation des strings** : Teste le support des chaînes de caractères et l'aptitude à les gérer (*longueur, contenu etc*)
    - **CE\_5 : Instruction nombres flottants** : Teste l'aptitude à raisonner sur des nombres flottants
- 
- UT\_3 : Exploration du Programme
    - **CE\_1 : Boucles** : Teste l'aptitude à gérer les boucles, les dérouler, etc
    - **CE\_2 : Chemins** : Teste l'aptitude à explorer les différents chemins d'un programme
- 
- UT\_4 : Découverte de bugs
    - **CE\_1 : Buffer-overflow** : Teste l'aptitude à détecter des débordements de tampon
    - **CE\_2 : Integer-overflow** : Teste l'aptitude à détecter des débordements d'en-tier
    - **CE\_3 : Off-by-One** : Teste l'aptitude à détecter des off-by-one (*réification des buffer-overflow*)
    - **CE\_4 : Use-After-Free** : Teste l'aptitude à détecter les Use-After-Free et similaires (*double-free*)
    - **CE\_5 : Format-string** : Teste la détection de format-strings (*contrôle de chaîne de format par l'utilisateur*)
    - **CE\_6 : Accès non autorisé** : Teste la détection d'accès (lecture/écriture) dans des zones mémoires non autorisées (*pas nécessairement non mappées*)
- 
- UT\_5 : Scale & Performances
    - **CE\_1 : Nombre de bugs trouvés** : Nombres de bugs trouvés dans les programmes “grande échelle”
    - **CE\_2 : Nombre chemin par seconde** : Nombre de chemins (*moyen*) testés par seconde dans le programme

- **CE\_3 : Exécutions par seconde** : Nombre d'exécutions (*moyennes*) du programme cible par seconde

Le terme symbolique, utilisé majoritairement dans un contexte de DSE, définit une variable *logique* sur laquelle un utilisateur a directement ou indirectement un impact sur sa valeur. Le contenu détaillé de chacun des CE est examiné dans la section 5.3 et 5.4 pour le passage à l'échelle.

Étant donné ces critères d'évaluation, différents aspects ont été écartés de la suite de test. Le premier est le test des différents mécanismes de teinte mis en œuvre dans certains outils. Tester de manière appropriée la teinte nécessiterait de tester la propagation au niveau bit et octets sur des instructions précises (décalage, extension de bit etc). Cela demanderait aussi d'effectuer des tests concernant la propagation de la teinte en mémoire avec des index symboliques. Le deuxième aspect éludé est le test de performance des solveurs qui est complètement hors périmètre de cet état de l'art. Enfin à propos du DSE, bien qu'il eut été possible de tester le support symbolique des différentes instructions, l'écriture des tests est très fastidieuse et ce genre d'information sera déjà récupérée lors de l'analyse fonctionnelle de chaque outil.

## 5.3 Tests atomiques

Les tests atomiques couvrent les unités de test UT\_1, UT\_2, UT\_3, UT\_4 et donc toutes leurs CE sous-jacentes. La suite des bombes logiques [370] a été utilisée comme point de départ, car elle couvre plusieurs des critères d'évaluations nécessaires pour le DSE. Cependant celle-ci a largement été modifiée, car rétrospectivement certaines catégories et certains tests se sont avérés peu pertinents, artificiels, voire même buggés. Un examen minutieux de certains tests et la manière dont ils ont été conçus témoignent malheureusement d'une faible connaissance du fonctionnement du DSE par leurs auteurs. Quelques-unes des modifications effectuées sur la suite logic-bombs sont listées ci-dessous :

- La catégorie “Parallel Program” a été supprimée, car aucun DSE, ni aucun fuzzer ou analyseur statique n'est en mesure d'analyser correctement des programmes concurrents.
- Certains tests utilisent extensivement la libc. Le succès du test n'est donc plus conditionné par l'élément à tester, mais par le support de la fonction de la libc par le moteur de DSE. Lorsque cela a été possible, les appels ont été remplacés par l'équivalent inliné.
- Certains tests ont été corrigés ou déplacés lorsqu'il n'étaient pas bien catégorisés. À titre d'exemple, le listing 2 contient de manière non voulue une vulnérabilité de Off-by-one de par la négligence des valeurs mises dans 11\_ary. En effet si  $x = 4$  11\_ary[x] vaudra 5 et donc une lecture hors tableau dans 12\_ary aura lieu. Initialement mis dans la catégorie accès mémoire symbolique celui-ci a été corrigé et une version légèrement modifiée toujours vulnérable a été mise dans UT\_4/CE\_3.

- La catégorie “Symbolic Variable Declaration” a été supprimée, car elle teste plutôt le support de fonctions de la libc que la déclaration de variables symboliques.
- La catégorie “Contextual Symbolic Value” a en partie été supprimée (*les tests intéressants ont été déplacés*). La catégorie “Covert Propagation” également.
- Certains tests comme celui donné en listing 3 ont été supprimés. Celui-ci est par exemple contestable à plusieurs égards. Tout d’abord `buf` ne sera jamais inférieur à 0, le compilateur optimise le code en supprimant le test qui retourne systématiquement 0. Ensuite le moteur de DSE devrait être en mesure de faire de la recherche de gadget automatiquement pour faire du Return Oriented Programming (ROP) et mettre l’adresse de `trigger` comme adresse pour écraser le `ret` avec le `strcpy`. Ce test s’apparente plus à de l’Automated Exploit Generation (AEG) et il est donc considéré hors périmètre.

---

**Listing 2** Exemple buggé de `stackarray_sm_12.c` de la suite `logic_bombs`)

---

```
1 int logic_bomb(char* s) {
2     int symvar = s[0] - 48;
3     int l1_ary[] = {1, 2, 3, 4, 5};
4     int l2_ary[] = {6, 7, 8, 9, 10};
5
6     int x = symvar%5;
7     if(l2_ary[l1_ary[x]] == 9){
8         return BOMB_ENDING;
9     }
10    else
11        return NORMAL_ENDING;
12 }
```

---

À partir de la base des bombes logiques modifiées et corrigées, les tests de la suite `program-verification-samples`<sup>27</sup> ont été ajoutés lorsque ceux-ci ne faisaient pas doublon avec un test existant. Subséquemment, de nouveaux tests ont été écrits pour compléter la base de tests. L’ensemble des tests est décrit dans les sections suivantes.

Note

Les bombes logiques sont initialement conçues pour le DSE, ainsi certaines bombes sont triviales à déclencher pour un fuzzer. Certains tests peuvent toutefois s’avérer difficiles à résoudre pour un fuzzer n’ayant aucune optimisation. De plus, cette suite prendra du sens pour les fuzzers lorsque ceux-ci seront combinés avec des moteurs de DSE.

---

27. <https://github.com/JonathanSalwan/program-verification-samples>

---

**Listing 3** Test stack\_bo\_12.c de la suite logic\_bombs

---

```

1 int trigger(){
2     return BOMB_ENDING;
3 }
4
5 int logic_bomb(char* symvar) {
6     char buf[8];
7     strcpy(buf, symvar);
8     if(buf < 0)
9         return trigger();
10    return NORMAL_ENDING;
11 }
```

---

### 5.3.1 UT\_1 : Calcul de prédicat de chemin et modélisation mémoire

Le calcul du prédicat de chemin pour le DSE ou d'une trace d'exécution pour le fuzzing se focalise sur les traitements effectués lors la rencontre de certaines constructions et en particulier les tableaux et la gestion de la mémoire. Le modèle mémoire est vraisemblablement l'un des points les plus difficiles à gérer pour du DSE (voir section 6.5). Les critères d'évaluations CE\_3, CE\_4 et CE\_5 se focalisent donc exclusivement sur les traitements effectués sur des données directement ou indirectement contrôlés par l'utilisateur et qui ont un impact sur l'activation de la bombe. Le tableau 5.1 présente tous les tests effectués pour UT\_1.

Les tests sur les tableaux (CE\_1) sont relativement simples et impliquent pour le fuzzer ou DSE de trouver la bonne valeur à un offset donné. L'exception est CId\_5 où la valeur lue vient d'un tableau non initialisé. Le fuzzer devra appliquer la force brute, tandis que le déclenchement dépendra du modèle mémoire pour un DSE. Soit il la considérera comme concrète et n'activera pas la bombe, soit il considérera qu'elle peut prendre n'importe quelle valeur et activera la bombe. Évidemment la reproductibilité du second n'est pas assurée.

Pour les index symboliques, un fuzzer devra muter les entrées pour activer de manière opportuniste la bombe. Un DSE, s'il possède un modèle mémoire symbolique (cf. 6.5), pourra les résoudre. En cas de modèle mémoire concret, il devra faire de la couverture de chemin sur toutes les valeurs possibles pour les adresses (*state-forking* cf. 6.5.2). Le nombre d'états possibles devient rapidement exponentiel. L'exemple CId\_16 stackarray\_sm\_rw\_11 donné en listing 4 présente le cas d'une écriture puis d'une lecture à un index symbolique. Un DSE avec un modèle mémoire symbolique pourra résoudre la bombe en un chemin via l'alignement du pointeur d'écriture et du pointeur de lecture.

---

28. Le trigger sous l'architecture ARM est différent : "x0l"

## Chapitre 5. Base de tests

	<b>Id</b>	<b>Nom</b>	<b>Description</b>	<b>LB</b>	<b>PS</b>	<b>New</b>	<b>Trigger</b>	<b>OV</b>
<b>CE_1 Tableaux</b>								
UT_1 : Calcul du prédictat de chemin et modélisation mémoire	CId_1	array_sample1	lecture index constant et valeur constante dans <code>argv</code>		✓		“AAA0xcc”	
	CId_2	array_sample2	ET logique lecture deux index constants dans <code>argv</code>		✓		“A0xaaA0xcc”	
	CId_3	array_sample3	copie de tableau puis lecture index constant		✓		* × 15+“H”	
	CId_4	array_sample4	plusieurs lecture index constants + test de différence		✓		“abcd”	
	CId_5	array_sample5	lecture index constant tableau non initialisé (.bss)			✓	“aléatoire”	
	CId_6	array_sample6	lecture index constant dans un tableau (.data)			✓	“a” × 12+“b”	
<b>CE_2 Propagation</b>								
	CId_7	file_cp_11	écriture puis lecture d'un fichier	✓			“7”	
	CId_8	file_posix_cp_11	écriture puis lecture d'un fichier (POSIX)	✓			“7”	
	CId_9	stack_cp_11	push puis pop dans une variable	✓			“7”	
<b>CE_3 : Lecture mémoire symbolique</b>								
	CId_10	stackarray_sm_11	lecture index symbolique sur la pile	✓			“4”	
	CId_11	stackarray_sm_12	double déréférencement avec index symbolique	✓			“2”	
	CId_12	stackarray_sm_ln	déréférencement recursif (n fois)	✓			“6”	
	CId_13	pointers_sj_11	lecture symbolique tableau de callbacks	✓			“5”	
<b>CE_4 : Ecriture mémoire symbolique</b>								
	CId_14	stackarray_sm_store_11	écriture symbolique puis lecture constante			✓	“Da”	
	CId_15	stackarray_sm_store_12	écriture symbolique hors bornes du tableau			✓	“Wol” <sup>28</sup>	✓
<b>CE_5 : Lecture/Ecriture mémoire symbolique</b>								
	CId_16	stackarray_sm_rw_11	écriture puis lecture symbolique dans un tableau			✓	“DDa”	

LB : Bombes Logique, PS : Program-Sample-Verification, New : Nouveaux tests ajoutés, OV : le test peut “crasher”

TABLE 5.1 – Suite de tests atomiques pour UT\_1

Un DSE avec un modèle concret et un fuzzer devront respectivement couvrir un espace de recherche de  $30^2$  et de  $2^8 \times 30^2$  possibilités.

### 5.3.2 UT\_2 : Symbolisation des entrées et modélisation des contraintes

La symbolisation des entrées pour le DSE et la gestion des entrées pour le fuzzing modulent le pouvoir de couverture dans le programme. Le fuzzing est plus dépendant du système, car dans le cas classique les entrées sous fournies uniquement à l'exécution du programme. Un DSE peut être amené à introduire de nouvelles valeurs symboliques sur des appels de fonctions externes (non exécutées) CE\_1 ou encore des instructions ou des registres ayant des valeurs non déterministes (CE\_3) lors de l'exécution. Cette unité de test regroupe aussi la gestion des contraintes sur les strings (CE\_4) et les nombres flottants CE\_5 (cf. tableau 5.2 ci-dessous).

CE\_1 et CE\_2 ne représentent pas de trop de difficultés pour un fuzzer. Un DSE en revanche doit exécuter symboliquement les fonctions de la libc ou bien les modéliser symboliquement. Dans le cas de l'instruction `syscall`, `svc` le DSE n'a pas d'autres choix que de modéliser l'appel système symboliquement ou bien de l'ignorer. D'un point de vue DSE `syscall` est donc une instruction pouvant avoir de nombreuses sémantiques différentes en fonction des paramètres. Pour les instructions non déterministes, CId\_30 teste le *TimeStamp Counter* qui s'incrémente de façon monotone. Le listing 5 montre le code spécifique à x86. Dans ce cas seul, la congruence à 2 est testée. L'équivalent en ARM est `pmccntr_el0` situé dans un coprocesseur et accessible via un `msr`. Bien qu'il soit

	<b>Id</b>	<b>Nom</b>	<b>Description</b>	<b>LB</b>	<b>PS</b>	<b>New</b>	<b>Trigger</b>	<b>OV</b>
<b>UT_2 : Symbolisation des entrées et modélisation des contraintes</b>								
CE_1 Fonctions externes								
CId_17	atoi_ef_12	test la fonction atoi	✓			“7”		
CId_18	pow_ef_12	test la fonction pow	✓			“7”		
CId_19	printint_int_11	test la fonction printf	✓			“7”		
CId_20	rand_ef_12	test la fonction rand avec argv en graine	✓			“aléatoire”		
CId_21	file_csv	ouvre un fichier ( <i>test contextuel</i> )	✓			nom fichier valide		
CId_22	file_posix_csv	ouvre un fichier (POSIX) ( <i>test contextuel</i> )	✓			nom fichier valide		
CId_23	pid_csv	test valeur de PID (modulo 5)	✓			“aléatoire”		
CId_24	malloc_1	test la fonction malloc et la copie		✓		toujours activé		
CId_25	memset_1	test la fonction memset		✓		“a” × 21		
CE_2 Appels système								
CId_26	syscall_alarm	test support syscall alarm et de la fonction de callback			✓	“1”		
CId_27	syscalls_open_close	test ouverture puis fermeture de fichiers			✓	toujours activé		
CId_28	syscalls_read_write	test écriture puis lecture du même fichier (+fseek)			✓	“a”		
CId_29	syscall_time	test syscall time			✓	“aléatoire”		
CE_3 : Instructions non-déterministes								
CId_30	rtdsc	test rdtsc sur x86			✓	“aléatoire”		
CId_31	cpuid_1	test le premier caractère du manufacturier			✓	tourner sur VMware		
CId_32	cpuid_2	test le nom complet du manufacturier avec argv			✓	nom manufactureur		
CE_4 : Modélisation des strings								
CId_33	string_sample1	test la fonction strcmp			✓	“trigger”		
CId_34	string_sample2	test la fonction strlen		✓		“aaa”		
CId_35	string_sample3	test longueur de argv( <i>meta-propriété</i> )			✓	* × 22		
CId_36	string_sample4	test nombre d'occurrence d'un caractère ( <i>meta-propriété</i> )			✓	chaîne dont 7 “A”		
CId_37	propagation_sample3	copie chaîne avec strncpy, strcmp	✓			“trigger”		
CE_5 : Instruction nombres flottants								
CId_38	simple_float1	test addition nombre flottants			✓	“9”		
CId_39	atof_ef_12	test la fonction atof		✓		“7”		
CId_41	sin_ef_12	test la fonction sin avec π		✓		“6”		
CId_42	ln_ef_12	test la fonction ln		✓		“7”		

LB : Bombes Logique, PS :Program-Sample-Verification, New : Nouveaux tests ajoutés, OV :le test peut “crasher”

TABLE 5.2 – Suite de tests atomiques pour UT\_2

**Listing 4** Test CIId\_16 stackarray\_sm\_rw\_l1

```
1 int entry(char *s) {
2     int symvar = s[0] - 48; //offset symbolique d'écriture
3     int symvar2 = s[1] - 48; //offset symbolique de lecture
4     int arr[30] = {0};
5
6     arr[symvar % 30] = s[2];
7
8     if (arr[symvar2 % 30] == 'a')
9         return TRIGGERED;
10    else
11        return NOT_TRIGGERED;
12 }
```

---

accessible en EL0, cela requiert de l'exposer en EL0 en modifiant le noyau<sup>29</sup>. Ne voulant aucun prérequis sur la configuration du noyau, ce test n'a pas pu être implémenté en ARM. `cpuid_1` et `cpuid_2` testent en revanche le nom fabricant du CPU accessible avec l'instruction `cpuid`. `CIId_31` test que le premier caractère est "V" ce qui n'est en principe vrai que pour les systèmes virtualisés par VMWare. Un fuzzer sera donc incapable d'activer la bombe, car il n'a aucune prise sur la valeur retournée dans les registres alors qu'un DSE en fonction de son fonctionnement concrétisera ou symbolisera la valeur. `CIId_32` effectue une comparaison du nom complet avec la chaîne en entrée un fuzzer et un DSE seront donc en mesure d'énumérer les bons caractères. `CIId_35` et `CIId_36` testent respectivement la longueur d'une chaîne et le nombre d'occurrences d'un caractère dans celle-ci. Il est difficile pour un DSE de raisonner sur ces deux "méta-propriétés" sur les strings. Il est en effet difficile pour un solveur de lier logiquement le nombre qui représente la taille avec le contenu lui-même (voir section 6.9). À noter que la plupart des DSE prennent un buffer symbolique de taille fixe. Soit celui-ci est suffisamment grand et la bombe pourra être activée soit celui-ci statuera sur l'infaisabilité du test.

**Attention**

Certaines bombes logiques et en particulier les tests d'instructions non déterministes renvoient sur ARM, la valeur `NOT_APPLICABLE`, car l'instruction n'existe pas ou l'équivalent nécessite d'être exécuté en mode privilégié (*exception-level 1 et plus*).

---

29. [http://zhiyisun.github.io/2016/03/02/How-to-Use-Performance-Monitor-Unit-\(PMU\)-of-64-bit-ARMv8-A-in-Linux.html](http://zhiyisun.github.io/2016/03/02/How-to-Use-Performance-Monitor-Unit-(PMU)-of-64-bit-ARMv8-A-in-Linux.html)

**Listing 5** Test CId\_30 rdtsc

```

1
2 int entry() {
3     int64_t rdtsc_val;
4 #if TARGET_X64
5     unsigned hi, lo;
6     __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
7     rdtsc_val = ((uint64_t)lo)|(((uint64_t)hi)<<32 );
8 #endif
9     if(rdtsc_val % 2 == 0)
10         return TRIGGERED;
11     else
12         return NOT_TRIGGERED;
13 }
```

**5.3.3 UT\_3 : Exploration du programme**

L’unité de test UT\_3 vise à évaluer la capacité d’exploration de l’outil testé. Cela implique d’être capable de raisonner sur les boucles, ce qui est testé par CE\_1. CE\_2 a pour but de tester la capacité de couverture sur des constructions parfois problématiques comme des switch (CId\_48, CId\_51, CId\_52) des “magic number” (CId\_49, CId\_50) ou sur des tableaux de fonctions (CId\_53). Le tableau 5.3 donne l’ensemble des cas de test de cette unité de test.

	<b>Id</b>	<b>Nom</b>	<b>Description</b>	<b>LB</b>	<b>PS</b>	<b>New</b>	<b>Trigger</b>	<b>OV</b>
<b>CE_1 : Boucles</b>								
UT_3 : Exploration du programme	CId_43	5n+1_lo_11	test un nombre de tour de boucle précis	✓			“7”	
	CId_44	collaz_lo_11	test la conjecture de Syracuse ( <i>Collatz</i> )	✓			“7”	
	CId_45	loop_sample1	“serial” avec xor sur la chaîne argv		✓		“elite”	
	CId_46	loop_sample2	calcule fonction de hash sur argv		✓		“elite”	
	CId_47	loop_sample3	“magic number” à casser avec atoi manuel		✓		“12345678”	
<b>CE_2 : Chemins</b>								
	CId_48	df2cf_cp_11	switch sur valeur de argv	✓			“7”	
	CId_49	sample_path1	“magic number” à casser		✓		“756234”	
	CId_50	sample_path2	“magic number” imbriqué		✓		“1234 823658”	
	CId_51	sample_path3	switch sur deux valeurs		✓		“54321”	
	CId_52	sample_path4	switch récursif		✓		“23857297”	
	CId_53	sample_path5	saut indirect via un tableau de fonctions		✓		“2”	
	CId_54	sample_path6	parseur d’expressions régulières ( <i>regex</i> )		✓		Hi-[1-9]{4}	

LB : Bombes Logique, PS :Program-Sample-Verification, New : Nouveaux tests ajoutés, OV :le test peut “crasher”

TABLE 5.3 – Suite de tests atomiques pour UT\_3

Le listing 6 donne un exemple classique de *serial* calculé à partir de la chaîne d’entrée.

**Listing 6** Test CIId\_45 loop\_sample1

---

```

1 char *serial = "\x31\x3e\x3d\x26\x31";
2
3 int entry(const char *s) {
4     int i = 0;
5     while (i < 5){
6         if (((s[i] - 1) ^ 0x55) != serial[i])
7             return NOT_TRIGGERED;
8         i++;
9     }
10    return TRIGGERED;
11 }
```

---

De manière générale, les tests de boucles représentent la même difficulté d'exploration pour un fuzzer ou pour un moteur de DSE. L'exception est CIId\_46 où toute la chaîne de caractères est systématiquement parcourue, ce qui permet théoriquement à un moteur de DSE de résoudre la contrainte en une exécution. Par ailleurs un fuzzer aura théoriquement plus de mal à résoudre CIId\_47 qui contient un *magic-number* correspondant à une constante 32 bits. Le dernier test significatif est CIId\_54 sample\_path6, issu d'un challenge. C'est une machine à état "accepteur" qui valide une chaîne si elle correspond à une expression régulière précise<sup>30</sup>. Le code de la fonction de validation illustré Figure 5.1 représente plus de 840 basic-block et permet donc de commencer à tester le passage à l'échelle en termes de couverture des chemins.

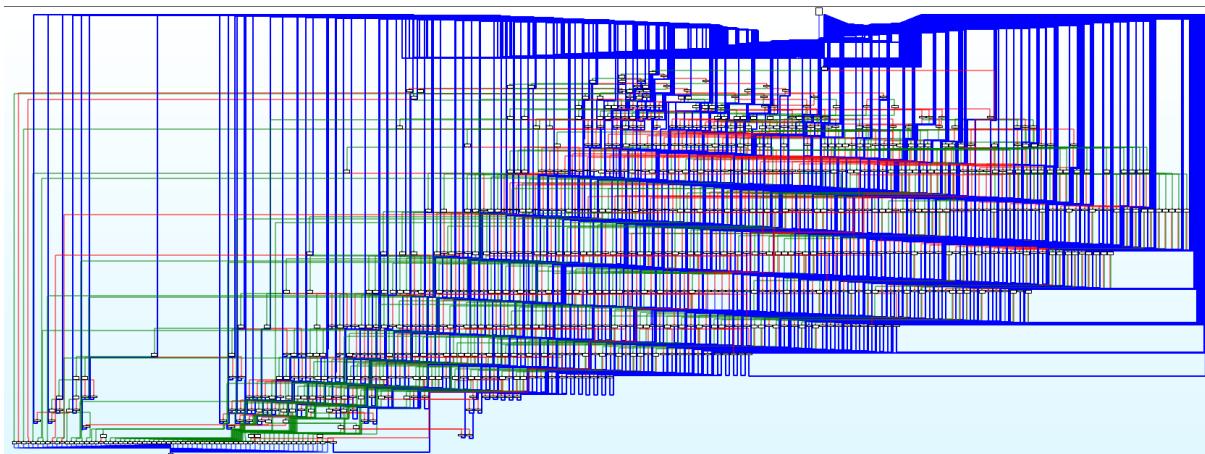


FIGURE 5.1 – Fonction validate\_input du test CIId\_54 : sample\_path6

---

30. [https://doar-e.github.io/blog/2013/08/24/regular-expressions-obfuscation\u2013under-the-microscope/#bring-out-the-fsm](https://doar-e.github.io/blog/2013/08/24/regular-expressions-obfuscation-under-the-microscope/#bring-out-the-fsm)

### 5.3.4 UT\_4 : Découverte de bugs

Celle-ci référence différents types de vulnérabilités, chacun d'eux ayant résulté en un critère d'évaluation dans la base de tests. Le tableau 5.4 présente les différents tests implémentés pour chacune des vulnérabilités.

UT_4 : Découverte de bugs	<b>Id</b>	<b>Nom</b>	<b>Description</b>	<b>LB</b>	<b>PS</b>	<b>New</b>	<b>Trigger</b>	<b>OV</b>
<b>CE_1 : Buffer-overflow</b>								
	CIId_55	b0f_sample1	BoF sur la pile sur <code>strcpy</code> ( <i>valeur quelconque</i> )		✓		* × 44	✓
	CIId_56	b0f_sample2	BoF sur la pile sur <code>strcpy</code> ( <i>valeur Oxdeadc0de</i> )		✓		* × 40 + 0xdec0adde <sup>31</sup>	✓
	CIId_57	stack_bo_11	BoF sur la pile ( <i>petit buffer</i> )	✓			"A" × 10 <sup>32</sup>	✓
<b>CE_2 : Integer-overflow</b>								
	CIId_58	ioF_sample1	IoF avec add	✓			"8"	
	CIId_59	ioF_sample2	IoF avec mul	✓			"? ? ?"	
	CIId_60	integer_overflow_1	IoF avec add et mul		✓		"9"	
<b>CE_3 : Off-by-One</b>								
	CIId_61	off_by_one_1	débordement sur <code>strncpy</code> ( <i>écrase une variable</i> )		✓		* × 9	
	CIId_62	off_by_one_2	débordement sur structure		✓		"9"	
	CIId_62	off_by_one_3	boucle for (erreur de ≤ au lieu de <)		✓		* × 8 + "A"	
	CIId_63	stackarray_sm_12_off_by_one	déréférencement symbolique hors tableau*		✓		"4"	
<b>CE_4 : Use-After-Free</b>								
	CIId_64	use_after_free_1	UaF sur déréférencement de fonction		✓		"H"	✓
	CIId_65	use_after_free_2	double <code>free</code> du même pointeur		✓		"H"	✓
<b>CE_5 : Format-string</b>								
	CIId_66	format_string_1	<code>snprintf</code> de <code>argv</code> en tant que format puis <code>strcmp</code>		✓		"%p"	
	CIId_67	format_string_2	<code>snprintf</code> transitif		✓		"%s"	
	CIId_68	format_string_3	écrasement de valeur de struct avec <code>sprintf</code>		✓		"%x%x"	
<b>CE_6 : Accès non autorisé</b>								
	CIId_69	read_anywhere_2	lecture adresse donnée via <code>argv</code>		✓		"aléatoire"	
	CIId_70	stackoutofbound_sm_12	"read anywhere" offset de 8 bits sur la pile		✓		"8"	✓

LB : Bombes Logique, PS : Program-Sample-Verification, New : Nouveaux tests ajoutés, OV : le test peut "crasher"

TABLE 5.4 – Suite de tests atomiques pour UT\_4

Aussi bien en DSE qu'en fuzzing, déclencher les bombes logiques est à la portée de la plupart des outils. Certains bugs sont protégés par de simples gardes comme CIId\_64, CIId\_65 pour lesquels l'intérêt n'est pas tant de trouver la vulnérabilité, mais plutôt de voir si celles-ci sont détectées. Le listing 7 montre l'exemple d'accès non autorisé ou pour valider la condition il est nécessaire d'aller chercher une valeur en dehors du tableau.

En conclusion, après avoir corrigé ou supprimé certains tests de `logic_bombs` puis intégré ceux de **program-verification-sample** 26 nouveaux tests ont été écrits. La suite de tests atomiques comptabilise 70 programmes qui permettent de balayer de manière précise les différents comportements des outils. Aucun critère d'évaluation n'a été conçu pour tester les communications réseau, car trop peu d'outils offrent cette fonctionnalité.

## 5.4 Test de passage à l'échelle

Le test de passage à l'échelle représente la deuxième partie de la suite de test et couvre l'unité de test UT\_5. Comme mentionné en section 4.4 de nombreuses suites de

31. Le trigger sous l'architecture ARM est différent : \* × 32 + 0xdec0adde

32. Le trigger sous l'architecture ARM est différent : "A" × 13

**Listing 7** Test CId\_70 stackoutbound\_sm\_12

```

1 int entry(char* s) {
2     int symvar = s[0] - 48;
3     int a[] = {1, 2, 3, 4, 5, 6};
4     if (a[symvar]<0 || a[symvar] > 6){
5         return TRIGGERED;
6     }
7     return NOT_TRIGGERED;
8 }
```

---

test sont à disposition pour tester le passage à l'échelle sur de vraies vulnérabilités et de vrais programmes. Un essai d'utilisation de la suite de Google `fuzzer-test-suite` a été fait. Nonobstant l'aspect réaliste des tests, le caractère spécifiquement conçu pour AFL et `libfuzzer` fut dissuadant quant à son utilisation. Le choix s'est tourné vers la suite LAVA-M largement utilisée dans le domaine du fuzzing. Cette suite se compose des binaires `uniq`, `base64`, `md5sum` et `who`. Le choix de ces binaires est discutable, et pour les tests seuls `uniq` et `base64` ont été retenus. En effet de précédents travaux internes [116] ont mis en évidence le manque de pertinence de `md5sum` et `who`. D'autre part le temps alloué au benchmark des outils ne permet pas de tester plus de deux programmes de manière rigoureuse. Le mode opératoire de test est décrit dans la section 5.5 suivante. Il eut été appréciable de tester le nombre de chemins ou le nombre d'exécutions par seconde des différents outils, mais l'hétérogénéité des outils rend la récupération de ce genre de métrique très fastidieuse. Ainsi, le seul critère d'évaluation retenu pour UT\_5 est : **CE\_1 : Nombre de bugs trouvés** : Définit le nombre de bugs trouvés par rapport au nombre total ayant été insérés dans chacun des deux programmes. D'autres métriques plus complexes ont été envisagées telles que le nombre d'exécution par seconde, le nombre de nouveaux chemin découverts par seconde ou encore des métriques basées sur la couverture. Cependant, en plus d'être particulièrement difficile à obtenir pour tous les outils elle ne sont pas nécessairement pertinentes de l'efficacité du passage à l'échelle et de l'efficacité de la recherche de vulnérabilité. Un exemple sur la métrique du nombre d'exécution par seconde est donné en annexe G.

## 5.5 Conditions & Environnement d'Exécution

### 5.5.1 Méthodologie & Protocole experimental de test

Les tests sont effectués sur des machines louées pour l'occasion et dédiées au projet. La première est un serveur x86 et la seconde un serveur ARMv8. Cela permet d'effectuer tous les tests sur les mêmes machines de référence.

Les caractéristiques de chaque machine sont décrites dans le tableau 5.5. Aussi bien le

	intel-x86.quarkslab.com	armageddon.quarkslab.com
<b>IP</b>	51.15.183.48	51.15.71.248
<b>OS</b>	Debian GNU/Linux 9 (Stretch)	Debian GNU/Linux 9 (Stretch)
<b>Noyau</b>	4.9.0-8	4.9.93
<b>Arch</b>	x86_64	ARMv8
<b>CPU</b>	Intel Xeon D-1531 @ 2.20GHz (6 cores)	ARMv8 QEMU @ 2GHz (6 cores)
<b>RAM</b>	32Go	4Go

TABLE 5.5 – Configuration des serveurs de tests

serveur x86 que le serveur ARMv8 correspondent aux caractéristiques techniques qu'auront respectivement l'ordinateur portable du démonstrateur et la carte ARM de test.

**Conditions de test** Les conditions de test varient entre les tests atomiques et les tests de passage à l'échelle. Les 70 tests atomiques sont exécutés **3 fois** pendant une durée maximum de **300 secondes**. Hors parallélisation cela représente un peu moins de 18 heures.

#### Note

Cette estimation est majorée. En effet, par exemple, certaines bombes logiques sont enclenchées facilement en quelques secondes par les fuzzers, permettant de réduire le temps total passé sur le benchmark.

Les deux tests de UT\_5 doivent eux être exécutés **3 fois** chacun, pour une durée de **6 heures**. Exécuter plusieurs fois permet d'obtenir des résultats plus robustes. En particulier pour le fuzzing, l'expérience montre que les graines et les différents aléas utilisés dans les outils peuvent introduire des écarts de résultats relativement significatifs [116] d'une exécution à l'autre. Effectuer 3 fois chaque test de UT\_5 permet donc de lisser l'aléa statistique propre au fuzzing.

**Paramétrage des outils** Chaque outil possède ses propres paramètres et ses propres optimisations. Il est donc quasiment impossible ni même souhaitable d'effectuer tous les tests sur le même pied d'égalité. La démarche choisie est d'activer tous les réglages et options qui permettent de maximiser les résultats. La seule exception est l'utilisation du parallélisme qui biaiserait trop les résultats. Le but est de mettre en évidence l'efficacité des optimisations et du fonctionnement et non pas la force de calcul pure.

---

#### Listing 8 Compilation de la suite de tests atomiques

---

```
mkdir build && cd build  
cmake -DTARGET_ARCH=[X64/AARCH64] [-DCMD\_LINE\_INPUT] ..  
make
```

---

### 5.5.2 Environnement de compilation

#### Compilation de la base de tests

La base de tests atomique fournie avec ce livrable se compile aisément grâce à `cmake`. Les deux paramètres à fournir sont l'architecture cible `X64` ou `AARCH64`, et spécifier si les entrées se font en ligne de commande et non pas sur `stdin` (`CMD_LINE_INPUT`). La compilation se fait donc avec la commande 8.

Évidemment, en fonction de l'outil à analyser, des adaptations sont à prévoir pour prendre en compte les spécificités de chacun. Les programmes de LAVA-M sont fournis au niveau source<sup>33</sup> et peuvent donc être facilement compilés aussi bien en `x86` qu'en `ARM`.

#### Note

Avec 70 tests pour la suite de tests atomiques et 72 bugs dans `base64` et `uniq` cumulés la suite de test totale (142 tests) se veut équilibrée quantitativement entre les bugs simples et les bugs de passage à l'échelle.

#### Configuration de la compilation

Lors d'une phase préliminaire de tests plusieurs problèmes sont survenus sur la suite des tests atomiques. Ceux-ci étant pour la plupart assez simples ou symptomatiques d'une construction de code à risque, les compilateurs ont tendance à être assez agressifs en termes d'optimisation et de réduction des risques (sur les constructions à risque). En conséquence il est apparu que certaines bombes logiques après compilation n'étaient pas activables soit à cause des optimisations soit à cause des comportements spécifiques des compilateurs.

**Optimisation aggressive** Après optimisation, le code de certaines bombes logiques était considéré comme du code mort et donc retiré du programme. Afin d'éviter toute optimisation, des directives spécifiques au compilateur ont été utilisées pour forcer la désactivation de l'optimisation. Certains compilateurs, comme `afl-gcc` ne semblent pas prendre en compte le paramètre de “non-optimisation” (`-O0`). Le Listing 9 montre les deux directives ajoutées sur le code chaque test atomique.

---

33. [http://panda.moyix.net/~moyix/lava\\_corpus.tar.xz](http://panda.moyix.net/~moyix/lava_corpus.tar.xz)

---

**Listing 9** Directive évitant l'optimisation

```

1 #ifdef __clang__
2     int __attribute__((optnone)) entry(char *s)
3 #else
4     int __attribute__((optimize("O0"))) entry(char *s)
5 #endif

```

---

**Listing 10** Cas de test CID\_58 : iof\_sample1

```

1 int entry(const char *str) {
2     int s = str[0] - 48 ;
3     if (s + 2147483640 < 0 && s > 0)
4         return TRIGGERED;
5     else
6         return NOT_TRIGGERED;
7 }

```

---

**Undefined-behavior.** La norme du C [189] laisse un certain nombre d'aspects du langage non définis que les compilateurs peuvent donc implémenter de manière différente. Ainsi, le débordement sur des entiers signés est un comportement indéfini dans la norme dont la sémantique diffère en fonction du compilateur voir de la version de celui-ci. L'exemple `iof_sample1` donné en Listing 10 donne un exemple de ce cas. L'entier signé `s` correspond au premier octet donné en paramètre auquel est ajouté 2147483640. Or, 2147483647 est la valeur maximale d'un entier sur 32 bits, la condition peut donc être validée en faisant déborder le résultat de l'addition.

Le problème rencontré et que `gcc 7` et `gcc 8` compile le programme avec une sémantique différente. La dernière version de `gcc` remplace l'entier signé par un entier non signé ne permettant plus à la bombe d'être activée. La figure 5.2 montre le code de la fonction compilé respectivement avec `gcc 7.4.0-6` et `gcc 8.3.0-6`. Dans le premier cas, l'addition est bien présente (`add eax, 7FFFFFF8`) alors que dans le second, le compilateur utilise directement une instruction de comparaison (`cmp [rbp+var_4], 80000008`). La solution est d'ajouter l'option `-fwrapv` à la chaîne de compilation de la suite de tests.

### 5.5.3 Reproductibilité

Le processus d'évaluation des fuzzers et des DSE possède toujours une part d'incertitude. En effet, les résultats obtenus peuvent différer d'un test à l'autre, sans changer aucun paramètre. En effet le processus d'évaluation n'est pas toujours déterministe, et peut varier, par exemple, en fonction de l'occupation des ressources disponibles, des mutations (en partie aléatoires) appliquées. Comme expliqué précédemment, afin de maximiser le déterminisme des benchmarks, le choix a été fait d'exécuter la base de tests trois fois

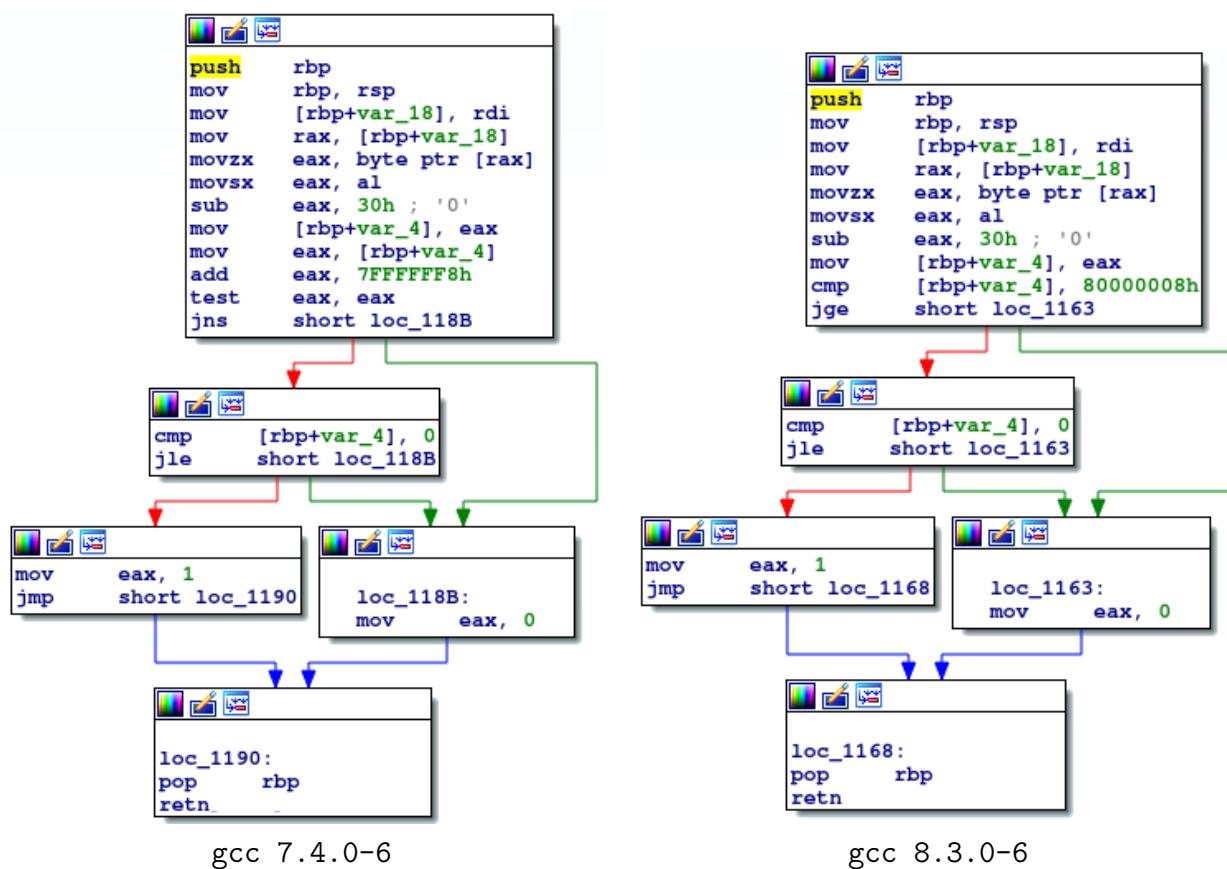


FIGURE 5.2 – Comparaison iof\_sample1 avec gcc 7 et 8

pour chaque outil.

Ces tests nécessitant un temps important, devant être exécutés plusieurs fois, un mécanisme d'automatisation de ces tests a été mis en place pour les **bombes logiques**. Cela permet aussi de facilement reproduire ces tests pour vérifier les résultats.

## Génération du fichier de configuration

Le script **samples-descr-gen** permet de générer un fichier de configuration pour les outils développés afin de benchmarker les différents fuzzers sur la suite des **bombes logiques**. Il génère un fichier **.yaml** et doit être utilisé comme suit :

---

```
$ samples-descr-gen --output samples.yaml concolic_test_suite/build/executables
```

---

Le fichier de configuration **samples.yaml** est alors généré. Celui-ci permet notamment aux outils de déterminer les chemins où sont situés les executables de tests.

**Listing 11** Exemple de fichier de configuration - **samples.yaml**

---

```
samples:
  5n+1_lo_11:
    base_path: './'
    binary_arguments: ''
    binary_path: concolic_test_suite/build/executables/CE1_Loop/5n+1_lo_11
    input_from_stdin: true
    input_path:
      - testcases
  7n+1_lo_11:
    base_path: './'
    binary_arguments: ''
    binary_path: concolic_test_suite/build/executables/CE1_Loop/7n+1_lo_11
    input_from_stdin: true
    input_path:
      - testcases
  array_sample1:
    base_path: './'
    binary_arguments: ''
    binary_path: concolic_test_suite/build/executables/CE2_Array/array_sample1
    input_from_stdin: true
  [...]
```

---

Une fois ce fichier de configuration généré, il est alors possible de démarrer la phase de tests grâce au script **fuzzer-benchmark** avec la commande donnée en Listing 12.

Les différents tests sont fuzzés de manière séquentielle. Cette opération est répétée 3 fois, et chacune des campagnes de fuzzing est sauvegardée dans le répertoire pointé par l'option **output-path /<# run>/benchmark.csv**. Une fois le processus terminé, la commande suivante permet de collecter les résultats :

**Listing 12** Exemple d'utilisation - fuzzer-benchmark

---

```
fuzzer-benchmark --fuzzer-name afl \
    --fuzzer-path /pastis/fuzzers/afl-2.52b/afl-fuzz \
    --runs 3 \
    --stop-on timeout=300,crash='sig:08' \
    --output benchmarks.csv \
    --input-path ./ \
    --output-path benchmarks/ \
    samples-all.yaml
```

---

```
cat benchmarks/*/benchmark.csv | sort > benchmarks.all.csv
```

---

Le fichier `benchmarks.all.csv` contient alors la liste de toutes les **bombes logiques** qui ont été déclenchées par le fuzzer en question. Ces fichiers peuvent être collectés pour les différentes exécutions des différents fuzzers, afin de comparer leurs résultats respectifs.

# **Troisième partie**

## **Exécution Concolique**



# Chapitre 6

---

## État de l'art : Éxecution Symbolique Dynamique

---

### 6.1 Introduction

L'exécution symbolique (SE) est une approche formelle fructueuse pour le test de code logiciel. Étant donné un chemin dans un programme, l'intuition clé est qu'il est possible dans la plupart des cas de calculer un *prédictat de chemin* tel qu'une solution à cette formule est un test (entrée) permettant d'exercer ce chemin. Ensuite, il ne reste plus qu'à explorer tous les chemins du programme pour le tester efficacement.

Les bases de cette technique ont été posées dans les années 70 par King [208], mais la technique a trouvé un regain d'intérêt dans les années 2000 [59] lorsqu'elle a été combinée avec de l'exécution concrète [311, 364] mais aussi grâce aux performances accrues des solveurs Satisfiability Modulo Theories (SMT). Le **Symbolic Execution (SE)** est rapidement devenu la technique la plus prometteuse pour la génération de tests automatisés et a rapidement mené à d'impressionnantes cas d'utilisations [16, 58] et une adoption au niveau industriel [139]. Son utilisation à des fins de sécurité a aussi trouvé plusieurs cas d'utilisation de par sa transposition relativement aisée sur du code binaire [326]. L'utilisation considérée dans cet état de l'art se focalise sur le **SE** pour la recherche de vulnérabilités [15, 170], qui a connu une incroyable impulsion en 2013 grâce au Cyber Grand Challenge organisé par la DARPA<sup>34</sup>. Cette compétition de deux ans avait pour but d'élaborer un **CRS** capable de détecter des vulnérabilités, les exploiter et les corriger en temps réel [320].

Traditionnellement SE désigne l'exécution symbolique *statique*, c'est-à-dire sans application des effets de bords du programme. Malheureusement, les interactions avec le système et l'environnement étant essentielles, il est souvent nécessaire de combiner le SE avec une exécution concrète. Cet enchevêtrement est habituellement désigné par **Dynamic Symbolic Execution (DSE)** ou encore *exécution concolique*, l'un et l'autre étant identiques. Se reposer sur une exécution concrète et donc des chemins d'exécution concrets, apporte les avantages suivants :

---

34. <https://www.darpa.mil/program/cyber-grand-challenge>

- de par l'aspect correct “*sound*” de l'exécution, le chemin est sûr d'être exerçable en pratique
- l'instruction suivante à exécuter est toujours connue. Cette propriété est particulièrement intéressante lorsque le programme comporte des sauts calculés ou des exceptions
- par défaut toutes les boucles sont déroulées.

Le principal inconvénient est que la trace suivie est une *sous-approximation* de l'exécution, car dépendant des entrées du programme. Les différentes approches sont détaillées en section 6.4.

La littérature regorge de travaux<sup>35</sup> sur le sujet du DSE. Cet état de l'art s'appuie sur un état de l'art de 2018 [19] et sur différentes ressources annexes [73, 92, 200, 284] présentées dans les sections ci-après. L'exécution symbolique soulève de nombreuses questions de recherche traitées et explorées par l'état de l'art actuel des publications. Les quatre problématiques sont la gestion mémoire (cf. section 6.5), la gestion des entrées symboliques et de l'environnement qui induisent du non-déterminisme (cf. section 6.6), l'explosion de l'espace des chemins (cf. section 6.7) et enfin la résolution de contraintes (cf. 6.8).

## 6.2 Définitions & Algorithme

### 6.2.1 Prédicat de chemin

Soit un programme  $P$  sur un vecteur de données d'entrées  $V$  prises sur un domaine de valeurs  $D$ . Un cas de test  $tcs$  pour  $P$  est une valuation de  $V$ , i.e.  $tcs \in D$  tel que l'exécution de  $P$  avec  $tcs$ , noté  $P(tcs)$ , est un chemin (ou trace) défini par :  $\pi \triangleq (l_1, \Sigma_1) \dots (l_n, \Sigma_n)$ . Dans ce contexte,  $l_i$  est un point de contrôle du programme (emplacement / adresse) de  $P$  et  $\Sigma_i$  dénote les états internes successifs de  $P$  (i.e. une valuation de toutes les variables globales, locales et de la mémoire) avant chaque exécution de  $l_i$ .

Intuitivement, un prédicat de chemin est la conjonction logique de tous les branchements logiques et de toutes les assignations rencontrées sur ce chemin. On note  $\varphi_\pi$  le prédicat logique associé au chemin  $\pi$ .

### 6.2.2 Algorithme d'exécution symbolique

En considérant l'ensemble fini des chemins d'un programme<sup>36</sup>  $P$  noté  $\Pi^P$ . Un algorithme symbolique construit itérativement un ensemble de tests explorant tous les chemins faisables via les trois composants suivants :

35. <https://github.com/enzet/symbolic-execution/blob/master/diagram/symbolic-execution.svg>

36. assuré en bornant la profondeur maximum d'un chemin

- $\text{Sel} : \Pi^P \rightarrow \pi$ , une stratégie de sélection de chemin, visant à choisir le chemin le plus approprié pour l'analyse à effectuer.
- $\mathbb{C} : \pi \rightarrow \varphi$ , la fonction de calcul du prédicat de chemin utilisant des théories (logique) prédéfinies notées  $T$  (cf. 6.8.3). Pour de la recherche de vulnérabilités, les bitvecteurs et les tableaux sont majoritairement utilisés.
- $\text{Sol} : \Phi \rightarrow \{\text{(SAT}, tcs) \cup (\text{UNSAT}, \emptyset)\}$  la fonction de satisfaisabilité utilisant un solveur automatique. Cette fonction prend une formule  $\phi \in \Phi$  telle que  $\Phi \subset T$  et renvoie soit SAT avec une valuation (solution)  $tcs$ , soit UNSAT sans solution.

À partir de ces fonctions, il est possible de définir l'algorithme d'exécution symbolique tel qu'illustré par l'algorithme 1.

---

**Algorithm 1** Algorithme d'exécution symbolique

**Require:** un programme  $P$  avec un ensemble fini de chemins  $\Pi(P)$

```

1:  $TS \leftarrow \emptyset$ 
2:  $S_{paths} \leftarrow \Pi(P)$ 
3: while  $S_{paths} \neq \emptyset$  do
4:    $\pi \leftarrow \text{Sel}(S_{paths})$ 
5:    $S_{paths} \leftarrow S_{paths} \setminus \{\pi\}$ 
6:    $\varphi_\pi \leftarrow \mathbb{C}(\pi)$ 
7:    $res \leftarrow \text{Sol}(\varphi_\pi)$ 
8:   if  $res = \text{sat}(tcs)$  then
9:      $TS \leftarrow TS \cup \{(tcs, \pi)\}$ 
10:  else                                      $\triangleright$  unsat
11:    skip
12:  end if
13: end while
      return  $TS$                           $\triangleright$  un ensemble de paires  $(tcs, \pi)$  tel que  $P(tcs)$  couvre  $\pi$ 

```

---

La fonction  $\text{Sol}$  est usuellement déléguée à un solveur SMT (voir section 6.8). L'intelligence d'un moteur d'exécution symbolique se concentre donc essentiellement dans  $\text{Sel}$  et  $\mathbb{C}$ . Par ailleurs, les problématiques posées par  $\text{Sel}$  sont relativement similaires à celles rencontrées en fuzzing pour la stratégie de sélection des configurations **Schedule** (voir section 9.4). Quant à elle, la fonction  $\mathbb{C}$  se repose généralement sur une représentation intermédiaire du langage analysé, à partir de laquelle une sémantique peut être exprimée en logique. La section suivante présente les propriétés des différentes représentations intermédiaires existantes.

## 6.3 Représentation intermédiaire

### 6.3.1 Introduction

La représentation intermédiaire est la pierre angulaire d'un moteur d'exécution symbolique, car l'exécution symbolique s'effectue sur ce langage. Une **Intermediate Representa-**

tion (IR) permet d'encoder des analyses agnostiques du langage source ou de l'architecture cible. La principale difficulté dans la conception d'une IR est de modéliser la sémantique du langage ou des instructions avec le plus de fidélité possible. L'utilisation d'une IR n'est pas propre à l'exécution symbolique et d'autres techniques d'analyse comme l'interprétation abstraite ou le calcul de plus faible précondition (WP) utilisent aussi une IR. Que l'IR soit utilisée pour de l'exécution symbolique ou non, cette section se focalise sur celles permettant d'analyser du code C,C++ sous sa forme source ou binaire.

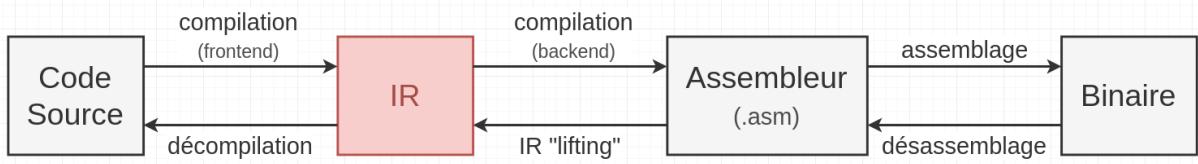


FIGURE 6.1 – Relation entre IR, source et binaire

Le schéma 6.1 montre le positionnement d'une IR dans une chaîne de compilation ou de décompilation. Certaines IR peuvent être obtenues depuis le langage source comme CIL [263] pour le C ou encore LLVM-IR qui peut être obtenue à partir de tous les langages ayant un frontend vers LLVM-IR. Autrement, elle peut être obtenue à partir du binaire comme pour VEX [264] ou REIL [112]. Ce type d'IR est généralement beaucoup plus bas niveau. À noter que LLVM-IR est la seule IR pouvant être obtenue aussi bien à partir des sources que du binaire [36].

---

```

res32 := (@[(esi(32) + 0x14(32))] ×(s) 7(32))
temp64 := ((exts @[(esi(32) + 0x14(32))] 64) ×(s) (exts 7(32) 64))
OF := (temp64(64) ≠ (exts res32(32) 64))
SF := ⊥
ZF := ⊥
CF := OF(1)
eax := res32(32)
  
```

---

Listing 6.1 – imul eax, dword ptr [esi + 0x14], 7

### 6.3.2 Représentations intermédiaires existantes

**Au niveau source** Les représentations intermédiaires dédiées au C, C++ sont très peu nombreuses de par la complexité de parser du code C et des différentes idiosyncrasies du langage à prendre en compte. CIL [263] spécifiquement dédiée au C, fournit un Abstract Syntax Tree (AST) d'un programme C avec une API en OCaml. Il est utilisé par Frama-C [209] comme pierre angulaire pour toutes les analyses effectuées dans le framework et en particulier Pathcrawler [364] effectuant du DSE. CIL est aussi utilisé par Otter [292] et dans un autre contexte par Tigress [84] pour de l'obfuscation. Ensuite vient LLVM-IR, utilisé dans LLVM comme langage intermédiaire entre les différents frontends et les différents backends. Il permet le support de quasiment tous les langages compilés et toutes les architectures processeur grand public. Depuis un code source, il est

simple d'obtenir le LLVM-IR associé avec des outils tel que `wl1vm`<sup>37</sup> (whole-program-llvm) qui, sans modification à un projet, permet de le compiler en LLVM-IR. Pour cela, l'outil fournit deux commandes, `wl1vm` et `extract-bc`, qui permettent respectivement de compiler le programme dans un '.a' et d'en extraire le bitcode. Depuis le binaire, l'opération est beaucoup plus complexe, car la compilation perd un certain nombre d'informations sur la structure ou le typage (cf. Section 6.3.3). D'autres IR génériques permettent la transformation de source en code natif. Un autre exemple est `Cranelift` IR [331], qui bien que centré sur WebAssembly, permet aussi de transformer l'IR vers du code natif.

**Au niveau binaire** Un grand nombre d'**IR** au niveau binaire existent. VEX, implémenté dans Valgrind [264], est sans doute le plus usité de par son très bon support de différentes architectures (x86, ARM, MIPS, PPC) et sa bonne modélisation des instructions complexes (flottants, etc). En conséquence, plusieurs outils utilisent VEX, comme `angr`, ou d'autres s'en servent pour dériver leur propre IR comme `Vine` [326], ou encore `BIL` [49] via la bibliothèque `libasmir`. À titre d'exemple une instruction x86 se décode en 4.69 instructions VEX [380]. REIL [112] fut l'une des premières IR résolument conçue pour le développement d'analyses binaires. Intégrée dans BinNavi, l'API permet d'implémenter toute sorte d'analyses en manipulant cette **IR** en Java (et plus particulièrement en Jython<sup>38</sup>). D'autres plateformes comme BARF [171], utilisent aussi cet **IR**. TCG [26] jouit d'un très bon support grâce à son utilisation pour de l'émulation dans QEMU. L'**IR** de P-code de Ghidra [266] publié très récemment est vraisemblablement l'une des plus évoluées avec le meilleur support d'architecture. En termes de rapidité de "lifting", LowUIR [197] récemment publiée semble être l'une des plus rapides avec des ordres de grandeur de x50 par rapport à des IRs lents tel que Miasm [255]. En x86, l'**IR** la plus complète est la plus exhaustive est sans contexte le `kframework` [95] qui sera présenté à la conférence PLDI le 26 juin 2019. Il modélise 7000 variantes d'instructions totalisant 774 mnémoniques dans un langage (appelé "k") créé à cet effet. Coté ARM, l'**IR** `Sail` [10] récemment publiée dans le cadre d'une collaboration entre ARM Ltd et l'université d'Oxford, fournit indiscutablement l'**IR** la plus aboutie pour ARM. À quelques instructions près, toutes les instructions jusqu'à ARMv8.3 sont modélisées. Cela inclut les exceptions, la translation d'adresses physiques/virtuelles et de nombreuses instructions privilégiées (EL1+). L'annexe C fournit une présentation plus détaillée de cette représentation intermédiaire pour l'**ISA** ARMv8.3.

Le tableau 6.1 résume les principales IR existantes utilisées aussi bien pour de l'exécution symbolique que de l'analyse statique au sens large. Pour chaque **IR**, il existe potentiellement plusieurs outils en plus de celui cité. Le tableau ne reflète pas la qualité du support de chaque architecture par les différentes IR, car il est souvent difficile d'obtenir le nombre d'instructions modélisées et si la modélisation est pertinente (cf. 6.3.4). À ce titre, LLVM-IR depuis le binaire est entièrement dépendant du "lifter" utilisé (cf. 6.3.3) et Vine ou BIL capitalisent sur le support des architectures par VEX. Il serait possible de comparer

37. <https://github.com/travitch/whole-program-llvm>

38. <http://www.jython.org/>

39. <https://github.com/Cr4sh/openreil/>

	Outils	Orig.		Architectures						FP	API	Transduct.
		Source	Binaire	x86	x86_64	ARMv7	ARMv8	MIPS	PPC			
BIL [49]	BAP	✓	✓	✓	✓	n/c	✓	✓	-	Oc,C,Py,R	n/c	
DBA [108]	Binsec	✓	✓	✓	✗	~	n/c	✗	✗	-	Oc	LLVM§
REIL [112]	BinNavi	✓	✓	✓	✓	n/c	n/c	n/c	n/c	-	J,Py‡,C‡	n/c
LLVM-IR [339]	KLEE	✓	✓	✓†	✓†	✓†	✓†	✓†	✓†	●	all	
VEX [264]	Valgrind	✓	✓	✓	✓	✓	✓	✓	✓	●	C,Py	BAP, REIL
Sail [10]	RMEM	✓	✓	✓	✓	✓✓	✓✓	✓	✗	●	Oc	Oc,C,Coq HOL4, Isa
CIL [263]	Frama-C	✓	-	-	-	-	-	-	-	●	Oc	-
Miasm IR [255]	Miasm	✓	✓	✓	✓	✓	✓	✓	✗	n/c	Py	LLVM
TCG [26]	QEMU	✓	✓	✓	✓	✓	✓	✓	✓	●	n/c	n/c
ESIL [286]	Radare2	✓	✓	✓	✓	✓	✓	✓	✓	●	n/c	n/c
Microcode [123]	Insight	✓	✓	✓	✗	✗	✗	✗	✗	-	C++	-
Sage III [237]	ROSE	✓	✓	✓	n/c	n/c	n/c	n/c	n/c	●	C++	-
Vine [326]	BitBlaze	✓	✓	✓	✓	✓	✓	✓	✓	●	C, Oc	-
Microcode [155]	IDA Pro	✓	✓	✓	✓	✓	✓	✓	✓	●	C++	-
LLIL [1]	Binary Ninja	✓	~	~	✓	✓	✓	✓	✓	n/c	C, Py	-
Cranelift IR [331]		✓	~	~	n/c	n/c	n/c	n/c	n/c	n/c	C++, R	LLVM
P-code [266]	Ghidra	✓	✓	✓	✓	✓	✓	✓	✓	✓	Java, Py	-
LowUIR [197]	B2R2	✓	✓	✓	✓	✓	✓	✓	✗	n/c	F#	-

§ : travaux en cours, Oc : OCaml, C : C, Py : Python, R : Rust, ‡ via OpenREIL<sup>39</sup>

† : dépend du lifter (cf. 6.3.3)

TABLE 6.1 – Tableau comparatif des représentations intermédiaires

les IRs sur d'autres critères comme la possibilité de modifier dynamiquement l'endianess, la gestion des valeurs non définies par la spécification, l'utilisation d'expressions dédiées comme `assume`, `assert` ou encore l'implication logique ( $\Rightarrow$ ).

#### Note

Toutes ces IR ont pour but d'encoder la sémantique des instructions, mais d'autres "IR" ont pour but de fournir un AST du programme binaire en reconstruisant le CFG et d'autres structures si nécessaire. C'est le cas de l'IR de GrammaTech GTIRB [149] et son désassembleur `ddisasm`<sup>a</sup> (qui utilise Datalog pour encoder plusieurs heuristiques de désassemblage). Cet IR est notamment conçu pour faciliter la réécriture de programme via ses fonctionnalités de rreassemblage.

a. <https://github.com/GrammaTech/ddisasm>

### 6.3.3 Lifter & Transducteurs d'IR

L'implémentation d'une IR étant particulièrement fastidieuse, certains outils utilisent une IR existante (ex : `angr` avec VEX) et d'autres convertissent des IR existante vers la leur (ex : BIL est généré à partir du VEX).

REIL, tel qu'implémenté dans OpenREIL<sup>40</sup>, traduit le BIL vers REIL ce qui implique la chaîne : Bin → VEX → BIL → REIL.

En sus, comme mentionné en annexe C, `Sail` permet de générer des émulateurs C, OCaml ou encore Coq<sup>41</sup>, à partir du `Sail`, ce qui rend le langage extrêmement polyvalent.

Pour LLVM, depuis le binaire l'opération est relativement délicate, car cet IR est beaucoup plus proche d'un langage haut niveau et requiert donc plus de traitement qu'un simple encodage de la sémantique d'une instruction.

Le lifting s'apparente presque à de la décompilation. Le premier à faire cela fut S2E [77] qui "lift" le x86 vers le LLVM-IR pour ensuite faire du SE avec KLEE. Pour cela, il se base sur TCG de QEMU en implémentant un backend qui transforme le TCG en LLVM-IR. Le moteur s'appelle Revgen<sup>42</sup>.

L'autre lifter, McSema [36], développé par Trail of Bits, effectue une extraction du CFG du programme via IDA Pro puis effectue la translation avec la bibliothèque Remill [37] qui contient le code de la translation vers LLVM-IR.

Il existe de nombreux autres lifters LLVM-IR dont la documentation de McSema fournit un tableau comparatif exhaustif<sup>43</sup>. Parmi ceux-ci, `llvm-mctoll`<sup>44</sup> développé par Microsoft

40. <https://github.com/Cr4sh/openreil/>

41. <https://coq.inria.fr/>

42. <http://s2e.systems/docs/Tutorials/Revgen/Revgen.html#using-revgen>

43. <https://github.com/trailofbits/mcsema#comparison-with-other-machine-code-to-llvm-bitcode-lifters>

44. <https://github.com/Microsoft/llvm-mctoll>

et `retdec`<sup>45</sup> développé par Avast – effectuant une décompilation complète – sont tous deux prometteurs.

### 6.3.4 Correction des IRs

Très peu de travaux s'attaquent au problème de la correction de la modélisation de la sémantique des instructions. Les travaux du Kaist [206], avec l'outil `MeanDiff`<sup>46</sup>, proposent une méthodologie d'évaluation des IRs en comparant les effets de bords attendus en les bisimulant (`bisimulation`) sur les mêmes instructions. Cela a notamment permis de trouver un total de 24 bugs de sémantique dans DBA, BIL et pyVEX.

### 6.3.5 Comparatifs entre IRs

Peu de travaux ont été effectués pour comparer l'efficacité du *lifting* des IRs. Encore une fois, les travaux du KAIST avec B2R2 [197], effectuent une analyse comparative de la rapidité de *lifting* de plusieurs IRs. À ce titre, le `ESIL` [286] de `Radare2` est particulièrement rapide tandis que `angr` [320] compte parmi les plus lents. Ces travaux décrivent plusieurs critères de comparaisons et plusieurs optimisations pouvant être faites sur l'IR.

### 6.3.6 Traitement de l'IR

Au-delà du strict support d'une quelconque architecture ou de l'exhaustivité des instructions supportées, la force d'une IR se trouve dans les raffinements ou les analyses qui lui sont appliquées pour la rendre plus proche d'une forme canonique ; les différents transducteurs pour la convertir dans un autre langage (cf. `Sail` avec le OCaml, C, Coq) ou encore tout l'écosystème d'outils qui l'utilise. À cet égard, LLVM-IR est incontestablement l'IR la plus aboutie avec tout l'écosystème construit autour d'elle. Une normalisation standard appliquée par plusieurs IR est la transformation en `Static Single Assignment` (SSA) qui fournit un certain nombre de facilités d'analyse. `Binsec` [108] fournit des fonctionnalités permettant de “lifter” les opérations bas niveau à base de masques utilisés sur les `cmp`, `jcc` en les remplaçant par les opérateurs plus haut niveau ( $\leq, <, \geq$  etc.) aussi supportés dans la théorie des bitvecteurs. Cela améliore la lisibilité de l'IR, mais pas forcément la solvabilité. À l'expérience, certains solveurs SMT raisonnent plus efficacement sur les opérations bas niveau à base de masques que sur les opérateurs plus haut niveau. De manière analogue `Miasm` [255] fourni différentes passes de raffinement<sup>47</sup> intégrant des opérateurs spécifiques (`CC_U<=`), des *Phi functions* ou encore un algorithme de SSA optimisé pour minimiser les variables intermédiaires [41].

---

45. <https://github.com/avast-tl/retdec>

46. <https://github.com/SoftSec-KAIST/MeanDiff>

47. [https://miasm.re/blog/2019/01/16/miasm\\_ir\\_getting\\_higher.html](https://miasm.re/blog/2019/01/16/miasm_ir_getting_higher.html)

## 6.4 Exécution symbolique dynamique

### 6.4.1 Introduction

L'exécution symbolique dynamique [137], aussi appelée “concolique”, est le fait de mélanger des valeurs concrètes à une exécution symbolique. De ce fait l'exécuteur symbolique tire parti des informations concrètes pour résoudre des contraintes. Inversement, l'exécuteur concret tire parti des nouvelles entrées générées pour couvrir de nouveaux chemins. En termes de fonctionnement, deux types de DSE se distinguent :

**online** : Un exécuteur *online* résout plusieurs chemins en une seule exécution en dupliquant son état mémoire pour chaque nouvelle branche à couvrir. Cela permet d'éviter de réexecuter certains préfixes de chemin. Cela implique néanmoins de conserver de nombreux états actifs en mémoire. Les outils supportant ce mode [15, 77, 320], dont KLEE [56], utilisent cette technique. Ce dernier utilise un mécanisme de *copy-on-write* pour conserver ses états. Les solveurs SMT peuvent aussi venir en aide grâce aux commandes *push* et *pop* (cf. section SMT 6.8) permettant d'ajouter et d'enlever des clauses à une formule sans tout résoudre à chaque fois. À noter que conserver plusieurs états en parallèle implique de conserver plusieurs instances concurrentes ayant potentiellement des effets de bords incompatibles entre elles.

**offline** : Un exécuteur *offline* traite chaque chemin indépendamment et fonctionne souvent en deux étapes. La première est l'exécution concrète avec génération de la trace d'exécution, puis la deuxième est l'exécution symbolique sur celle-ci. La consommation mémoire est donc plus faible que pour le mode *online*. SAGE [139] et Binsec [99] fonctionnent de cette manière.

L'outil Mayhem [64] utilise un mécanisme hybride alternant entre rapidité et mémoire. Pour cela il commence en mode *online* et utilise un mécanisme de *checkpoint* lorsque le nombre de configurations actives (états) atteint un seuil. Un *checkpoint* permet de suspendre l'exécution du programme et de sauvegarder l'état du chemin courant sans avoir à relancer une exécution complète. Quelque soit le type d'exécuteur symbolique, le DSE fait intervenir une instrumentation dynamique du programme (cf. 6.4.3) et des techniques de mélange de valeurs concrètes/symboliques (cf. 6.4.2) qui impactent nécessairement le calcul de prédicat de chemin du SE classique.

### 6.4.2 Concrétisations & Symbolisations

Au-delà d'une approche purement symbolique, les outils de DSE récents tirent parti de leur aptitude à n'évaluer symboliquement qu'un sous-fragment d'intérêt dans une trace. Pour cela, deux mécanismes sont disponibles : la *concrétisation* notée  $\mathcal{C}$  et la *symbolisation* notée  $\mathcal{S}$ . La concrétisation utilise les valeurs obtenues lors de l'exécution pour *sous-approximer* le prédicat de chemin alors que la symbolisation va à l'inverse *sur approximer* celui-ci (*en introduisant de nouveaux symboles logiques*). Le premier permet

notamment de prendre en charge des effets de bords non modélisables ou trop coûteux calculatoirement (haché cryptographiques), alors que le deuxième permet notamment de généraliser des effets de bord inconnus. Le comportement par défaut appelé *propagation* noté  $\mathcal{P}$ , correspond au calcul du prédictat de chemin logique sans aucune autre approximation (modélisation parfaite). On note  $\rho \triangleq \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$  l'ensemble des choix possibles pouvant être faits par un exécuteur symbolique sur n'importe quelle expression durant l'exécution.

La figure 6.2 présente un programme simple avec des assignations combinées à un assert  $x > 10$  et les trois prédictats de chemins différents possibles pour le même chemin. Il est facile de constater que  $\varphi_1$  est correct et complet, car il correspond à une propagation parfaite. À l'inverse, en décidant de concrétiser  $a$ ,  $\varphi_2$  reste correct, mais devient incomplet. Enfin pour  $\varphi_3$ , en décidant de symboliser l'opération par un nouveau symbole non contraint (*fresh*), la complétude est préservée, mais la correction est perdue, car la propriété  $x_1 = a \times b$  n'est plus assurée par  $\varphi_3$ .

program	$\varphi_1$ : propagation ( $\mathcal{P}$ )	$\varphi_2$ : concrétisation $\mathcal{C}$	$\varphi_3$ : symbolisation $\mathcal{S}$
inputs : a, b		$a = 5$	
$x := a \times b$	$x_1 = a \times b$	$x_1 = 5 \times b$	$x_1 = \text{fresh}$
$x := x + 1$	$\wedge x_2 = x_1 + 1$	$\wedge x_2 = x_1 + 1$	$\wedge x_2 = x_1 + 1$
//assert $x > 10$	$\wedge x_2 > 10$	$\wedge x_2 > 10$	$\wedge x_2 > 10$

FIGURE 6.2 – Path predicate, concretization and symbolization

Le choix des concrétisations et des symbolisations abbréviées (C/S) est un élément crucial des moteurs de SE. Il a un impact direct sur le calcul de prédictat de chemin [274]. Regrettablement, très peu d'études ont tenté d'étudier l'impact des C/S sur les performances et la pertinence des résultats. Certains outils comme S2E [77] proposent un mécanisme d'annotation manuel, d'autres comme Binsec [98] proposent un mécanisme de politiques permettant de spécifier le comportement attendu en termes de C/S pour n'importe quelle expression logique.

### 6.4.3 Sources de valeurs concrètes & Instrumentation dynamique

Ce qui différencie un SE d'un DSE est que ce dernier s'appuie sur une source de valeurs concrètes quelle qu'en soit l'origine. Les outils de DSE peuvent avoir deux approches du terme “concrétisation” :

- pour les DSE “émulant” tout, la concrétisation est le fait de réduire une variable symbolique à une seule valeur constante (généralement déterminée avec un solveur SMT).
- pour les DSE basés sur une exécution concrète, la concrétisation est le fait de récupérer la valeur concrète (du registre ou cellule mémoire) dans le contexte concret qui peut être un GDB, une trace d'exécution concrète ou toute autre source d'Instrumentation dynamique.

Cette deuxième approche nécessite d'exécuter réellement le programme sur le système pour profiter de tous les effets non déterministes de l'environnement et donc de toutes les valeurs concrètes d'exécution (sans avoir à les modéliser symboliquement). Certains DSE ont un fonctionnement hybride, car ils n'exécutent pas tous le programme sur le processeur, mais seulement les appels pouvant produire des effets de bords (grâce à du *Just-In-Time (JIT)*). Comme vu précédemment le mode “online” (cf. 6.4.1) est nécessairement plus dur à implémenter, car il requiert d'être capable d'interrompre les threads, de restaurer des contextes et éventuellement des portions mémoire s'il est possible de revenir en arrière sur le chemin. De surcroît, gérer plusieurs chemins en parallèle implique de gérer les effets de bords concurrents sur le système. S2E [77] tire parti de la virtualisation de QEMU pour empêcher la propagation d'effets de bords de l'exécution d'un chemin à un autre. Cela permet notamment d'émuler tout l'OS avec le système de fichier etc. Le tableau 9.1 de la partie fuzzing mentionne la plupart d'entre eux. En sus, des outils tels qu'Avatar<sup>2</sup> [258] poussent la logique plus loin encore en fournissant un mécanisme d'orchestration entre analyses dynamiques via une interface uniformisée. Cela permet en théorie de changer le backend d'analyse dynamique de façon interchangeable. angr s'en sert pour effectuer son DSE sélectif (voir ci-dessous 6.4.4).

#### 6.4.4 Calcul du prédicat d'un chemin

Le calcul du prédicat de chemin  $\varphi$  associé au chemin  $\pi$  correspond à la transformation d'une trace d'exécution vers un prédicat logique  $\phi \in \Phi$  dans une théorie  $T$  prêt à être résolu par un solveur.

**Définition 2.** soit  $\varphi$  un prédicat de chemin pour  $\pi$  et  $tcs \in D$ , des valuations des entrées. On dit que  $tcs$  satisfait  $\varphi$  si et seulement si  $P(tcs)$  couvre  $\pi$ .

Pour rappel  $\varphi$  est dit **correct** si toute valuation de la formule  $\varphi$  couvre  $\pi$ . Il est dit **complet** si toutes entrées couvrant  $\pi$  est une solution à  $\varphi$ . Tiré d'une trace d'exécution concrète ; le  $\varphi$  est théoriquement correct et satisfiable (SAT). Néanmoins, de par les différentes approximations de modélisation qu'un moteur est obligé de faire, la formule n'est pas toujours conforme au chemin concret emprunté (cf. section 6.6).

**Selective Symbolic Execution** est une approche alternative de calcul de prédicat de chemins utilisée par S2E [77] basée sur l'observation que l'on peut vouloir explorer un composant (une fonction ou autre) sans vouloir se préoccuper du reste. Elle joue pour cela sur les concrétiisations mentionnées dans la section 6.4.2 précédente. Cela permet aussi d'exécuter concrètement la plupart des appels de bibliothèques sans avoir à les modéliser symboliquement (cf. section 6.6.2). Pareillement, angr [320] a récemment introduit des fonctionnalités s'apparentant à de l'exécution symbolique sélective<sup>48</sup>. Pour cela il utilise le concept de **ConcreteTarget** sur laquelle il est possible de “switcher” pour exécuter concrètement le programme afin d'en récupérer le contexte et recommencer ensuite de l'exécution symbolique.

---

48. [http://angr.io/blog/angr\\_symbion/](http://angr.io/blog/angr_symbion/)

### 6.4.5 Optimisations

Une optimisation pouvant être effectuée sur le chemin est une analyse de teinte [302, 307]. Cette analyse de flot de données vise instinctivement à teindre les entrées d'un programme, effectuer la propagation et regarder quelles sont les sorties toujours teintées. Des outils comme `Mayhem` [64] utilisent la teinte pour trouver les instructions de saut dont l'adresse est teintée. Une étude complète des mécanismes de calcul de teinte et de leur impact sur le DSE/Fuzzing est laissée hors périmètre de cet état de l'art.

### 6.4.6 Requête sur un chemin & Exploitabilité

Le processus montré précédemment, présente l'algorithme de DSE général qui, pour chaque chemin calcule le prédicat associé et le résoud par SMT. Cette approche est le comportement nominal préalable à toute autre requête, mais il faut distinguer les requêtes qui permettent d'améliorer la couverture des requêtes qui permettent de résoudre la problématique de l'analyse. En fonction de la problématique, pour un même chemin plusieurs requêtes SMT peuvent donc être effectuées. Les requêtes peuvent tester des expressions issues d'assertions, des pré-post conditions ou encore la présense de bugs. Un DSE a pour fonction de résoudre des problèmes de couverture donc juste de l'atteignabilité. Afin de détecter des bugs, ou, plus difficile, de tester leur exploitabilité, de nouvelles requêtes doivent être faites. Pour cette problématique trois types de requêtes sont nécessaires :

- requêtes d'atteignabilité (le prédicat de chemin est-il satisfiable ?)
- requête de détection de bug (e.g., le pointeur peut-il être détourné)
- requête d'exploitabilité (e.g, le pointeur peut-il pointer vers une zone teintée de la mémoire)

Tandis que la formule, du prédicat de chemin est naturellement calculée par le DSE, encoder une formule correspondant à l'exploitation d'une vulnérabilité est bien moins triviale. Elle nécessite de savoir comment modéliser une vulnérabilité et l'exploit qui va avec. `Mayhem` [64] conserve une teinte de la mémoire et génère des exploits correspondant au détournement d'un pointeur vers une zone contrôlée par l'utilisateur (cf. Section 7.2).

## 6.5 Modélisation de la mémoire

### 6.5.1 Introduction

La gestion de la mémoire est sans nul doute l'aspect le plus délicat et le plus dur rencontré dans le contexte du DSE [347]. Représenter les accès mémoires au niveau sémantique des instructions est relativement aisé (cf. section 6.3 IR) et ceux-ci se traduisent facilement en logique notamment avec la théorie des tableaux SMT [130]. Malheureusement, la complexité des formules générées est difficilement surmontable. Les moteurs de SE les traitent donc différemment en faisant un équilibre entre correction et complétude.

**Listing 13** Exemple simple accès symbolique [19]

```

1 void test(unsigned int i, unsigned int j) {
2     int a[2] = {0};
3     if (i > 1 || j > 1)
4         return;
5     a[i] = 5;
6     assert(a[j] == 5);
7 }
```

Certains outils considèrent toutes les lectures/écritures en mémoire comme étant concrètes et ne pourront alors pas raisonner de manière symbolique sur la valeur de ces pointeurs. L'exemple jouet 13 illustre ce problème. Pour prouver la validité de l'assert `a[j] != 5`, le moteur devra symboliquement assigner la même valeur à `i` et `j` pour être sûr que la valeur lue sera bien celle écrite juste avant. Avec des index concrets, le SE renverra SAT si “par chance” les deux valeurs concrètes étaient identiques, et UNSAT sinon. Pour activer l'assert, il devra alors itérer la combinatoire de toutes les valeurs possibles des deux entiers, à savoir 4 sans débordement et  $2^{128}$  si l'on considère des entiers 64 bits. La problématique de recherche qui se pose est :

**RQ 1.** *Comment traiter la mémoire, l'aliasing de pointeurs, le stockage de données à des adresses symboliques et comment modéliser la segmentation mémoire en vigueur sur un système d'exploitation ?*

### 6.5.2 Stratégies de couverture des valeurs de pointeurs

La gestion des pointeurs mémoire est le point le plus critique pour un passage à l'échelle du DSE [347]. L'approche la plus coûteuse en difficulté de résolution, mais la plus expressive d'un point de vue logique est de **propager** tous les accès (lecture/écriture) symboliques. Ainsi, l'opération est propagée sans approximation. Dans l'exemple 13, la contrainte `a[i] == 5` sera ajoutée dans le prédictat de chemin. Dans le cas où les index sont considérés comme concrets, deux approches ont été proposées pour permettre de couvrir les différentes valeurs d'adresses [208] :

**state-forking** : Au même titre que l'état courant est “forké” sur une opération de branchement dans le CFG, l'état est “forké” pour chaque valeur de lecture ou écriture en mémoire. Dans l'exemple 13 (sans considérer d'overflow), l'état serait dupliqué avec dans l'un la contrainte `a[0] == 5 ∧ i == 0`, et dans l'autre `a[1] == 5 ∧ i == 1`. Dans les deux cas, cela revient à concrétiser l'adresse d'écriture.

**if-then-else** : tire parti de la capacité de certains solveurs SMT à supporter l'expression logique `ite` dont le type est :  $ite : \text{bool} \times \mathcal{E}\text{xp} \times \mathcal{E}\text{xp} \rightarrow \mathcal{E}\text{xp}$ . Dans l'exemple 13, il est alors possible de représenter logiquement les deux possibilités sans avoir à “forker” les états. Cela se fait avec la contrainte : `a[0] == (ite, i == 0, 5, 0) ∧ a[1] ==`

( $ite, i == 1, 5, 0$ ). Bien que cela n'introduise pas de “fork”, l'ajout de *disjonctions* dans une formule est coûteux en performance. Par ailleurs, cela revient logiquement aussi à concrétiser l'adresse d'écriture.

Plusieurs outils dont EXE [57], KLEE [56], SAGE [139] utilisent une représentation purement symbolique de la mémoire et donc “propagent” simplement en présence d'opérations symboliques. D'autres outils comme CUTE [311], DART [137], Triton [304] ont une représentation concrète des tableaux. DART applique du random-testing sur les valeurs possibles des pointeurs. Cela réduit considérablement le pouvoir de décision de l'exécuteur symbolique et implique de nombreux faux négatifs. Néanmoins, c'est à ce jour la seule méthode permettant de réellement passer à l'échelle sur de gros chemins.

#### Note

Avec une gestion purement concrète des lectures/écritures, il est possible de se passer de la théorie des tableaux SMT en remplaçant chaque cellule mémoire “libre” (symbolique) par un bitvecteur (cf. section 6.8.5). Cela permet d'utiliser des solveurs ne supportant pas, ou mal, la théorie des tableaux.

### 6.5.3 Procédures de décision

Des outils tels que Mayhem [64] ou Binsec [99] introduisent une certaine modularité dans la gestion des lectures/écritures symboliques. Mayhem fut le premier à proposer une procédure de décision pour la mémoire. Il concrétise toutes les écritures et les lectures restent symboliques si l'ensemble des valeurs possibles calculé avec un Value Set Analysis (VSA) [18] ne dépasse pas 1024.

angr propose plusieurs modes de concrétisations dont un mode complètement concret, un mode qui concrétise les écritures et conserve symboliques les lectures dont les valeurs sont comprises dans un petit intervalle contigu de 128 et enfin un mode complètement symbolique. Il est néanmoins possible d'implémenter n'importe quelle stratégie grâce à son API<sup>49</sup>. MemSight [87] se présente comme un plugin de angr fournissant un modèle mémoire plus symbolique avec un mécanisme d'intervalle de valeurs appelé *paged interval tree*, associé à chaque adresse symbolique. Sans révolutionner la gestion de la mémoire, cette technique semble fournir de meilleurs résultats que les autres modes de angr.

Enfin Binsec propose un mécanisme de politiques de concrétisation et de symbolisations abrégé C/S [98]. Une politique permet de définir (la granularité allant jusqu'à l'instruction) quelle action effectuer ( $\mathcal{C}, \mathcal{S}, \mathcal{P}$ ) sur celle-ci pour chaque lecture/écriture dans le programme. Ces politiques sont principalement utilisées pour moduler les opérations mémoires, mais elles s'appliquent sur n'importe quelle expression du programme.

---

49. [https://docs.angr.io/advanced-topics/concretization\\_strategies](https://docs.angr.io/advanced-topics/concretization_strategies)

Une fois les C/S effectuées et la formule générée, il est encore possible de réduire la difficulté des formules en effectuant quelques optimisations sur les tableaux [119] (cf. section 6.8.5 optimisations SMT).

## 6.6 Gestion des entrées et modélisations symboliques

Cette section aborde les traitements effectués sur toutes les entrées *inputs* d'un programme. Beaucoup d'outils de DSE ou de fuzzing considèrent comme seules et uniques entrées `argv` et `stdin` d'un programme. Nonobstant les facilités que cela apporte, les entrées d'un programme sont bien plus nombreuses et complexes. En effet, une grande partie de la correction de l'exécution dépend de code non tracé par l'exécution ou d'événements dûs à l'environnement. Oublier de prendre en compte les données symboliques issues de l'interaction avec le système affecte inévitablement la pertinence de l'analyse.

### 6.6.1 Problématique

La problématique posée par la gestion des inputs et la modélisation des effets de bord est qu'un prédicat de chemin peut ne pas refléter l'exécution concrète sur lequel il se base et introduire une décohérence.

**Définition 3.** *La décohérence, ou perte de correction d'un prédicat de chemin, est le phénomène par lequel une valeur logique est rendue incorrecte vis-à-vis de l'exécution du programme à cause d'un effet de bord non pris en compte dans l'exécution originale.*

On distingue deux cas de décohérence. Le premier, **sat-decoherence** est une décohérence brisant la correction de l'exécution tout en préservant la satisfiabilité du prédicat de chemin. Le deuxième, **unsat-decoherence** brise la correction et rend  $\varphi$  UNSAT. Ce dernier est évidemment plus simple à détecter. Un sous cas de la décohérence SAT est appelé divergence de chemin (*path divergence*).

**Définition 4.** *La divergence de chemin a lieu lorsque les entrées générées pour un chemin mènent à l'exécution d'un autre chemin. Ceci est principalement dû à une mauvaise ou imprécise modélisation d'effets de bords.*

Un chemin incohérent UNSAT est un *faux négatif*, tandis qu'un chemin SAT dont les entrées générées provoquent une divergence est un *faux positif*. Certaines recherches ont montré que le taux de faux positifs pouvait atteindre 60% [139]. La référence [72] identifie les appels externes, les exceptions, les “casts” et les accès mémoires symboliques comme étant les principaux facteurs.

**RQ 2.** *Comment traiter les interactions avec l'environnement (appels de bibliothèques, appels systèmes ou tout autre événement) apportant du non-déterminisme dans l'exécution ? Tester tous les effets de bord possibles est évidemment impossible alors comment les approximer en maximisant la correction et la complétude tout en minimisant la décohérence et la divergence ?*

## 6.6.2 Gestion des appels de bibliothèques

Modéliser l'environnement système et les appels de fonctions de bibliothèques (*en particulier la libc*) est particulièrement ardu. Les outils comme DART [137], EXE [57] ou CUTE [311] exécutent l'appel avec les arguments concret. Cela limite les comportements du programme explorables, mais conserve la correction. AEG [15] supporte 70 appels de bibliothèques et appels systèmes, dont certains pour la gestion des threads et des processus. Dans `angr`, la gestion symbolique d'un appel de bibliothèque s'appelle `SimProcedure`, et différentes `SimProcedure` peuvent être implémentées grâce à l'API<sup>50</sup>. Pour la plupart des appels de la libc, KLEE se base sur `uClibc`,<sup>51</sup> une libc embarquée minimaliste beaucoup plus légère que `glibc`. Cela permet de rentrer dans la plupart des appels de la libc et les exécuter symboliquement pour un surcoût limité. Par ailleurs, KLEE est capable de créer un système de fichier symbolique grâce à l'option `-sym-files` permettant de symboliser un fichier<sup>52</sup>. KLEE utilise un traitement analogue pour les sockets en symbolisant leur contenu. Cloud9<sup>53</sup> [51, 52], basé sur KLEE ajoute le support de plusieurs fonctions POSIX et un meilleur contrôle de l'environnement de test. Mais surtout, en ce qui concerne le réseau, Cloud9 permet de tester la fragmentation, la perte et le réordonnancement de paquets. Il est le seul à gérer ce type de test au sein d'un moteur d'exécution symbolique.

## 6.6.3 Gestion des appels système

D'un point de vue SE, la gestion des appels système est analogue aux appels de bibliothèques à la différence que le moteur n'a pas d'autre choix que d'ignorer l'appel ou de le modéliser. Il n'est pas possible de "rentrer" dans l'appel système pour l'exécuter symboliquement. À la connaissance des auteurs, les seuls moteurs capables d'entrer dans les appels système pour continuer l'exécution symbolique sont Reven [340] de Tetrane pour le x86 et potentiellement l'IR Sail [10] pour ARM de par sa modélisation sémantique des instructions EL1+.

## 6.6.4 Gestion des instructions non déterministes

De la même manière que pour les appels de bibliothèque et encore plus des appels système, certaines instructions ont un comportement non déterministe. Lors du DSE, il convient donc d'appliquer les concrétisations ou les symbolisations nécessaires pour ne pas induire de décohérence (SAT ou UNSAT) du prédicat de chemin généré. Les deux instructions connues pour cela en x86 sont `cpuid` et `rdtsc`. Sur ARM certains registres sont non déterministes et l'appel à des coprocesseurs avec `mrs` est aussi une forte source de non-déterminisme. L'exemple 14 montre une lecture du nombre de cycles du processeur via cette instruction. Certaines représentations intermédiaires possèdent une expression

---

50. <https://docs.angr.io/extending-angr/simprocedures>

51. <https://github.com/klee/klee-uclibc>

52. <https://klee.github.io/tutorials/using-symbolic/>

53. <http://cloud9.epfl.ch/>

**Listing 14** Exemple d'instruction non déterministe `mrs`

```

1 uint64_t get_processor_cycle_count() {
2     uint32_t pmccntr32;
3     asm volatile("mrs %0, pmccntr_el0" : "=r" (pmccntr32));
4     return (uint64_t) pmccntr32;
5 }
```

du type `nondet`, ou  $\top$  représentant une valeur inconnue. Cela permet de toujours obtenir le même IR pour la même instruction. Le choix de l'action à effectuer est laissé à l'analyse subséquente (DSE, interprétation abstraite, `WP` etc).

**argv variadique**

L'aspect variadique de `argv` est problématique pour de nombreux moteurs de SE car la longueur est souvent spécifiée par l'utilisateur. Le moteur va alors symboliser toute la chaîne et considérer le reste comme concret ou incontrôlable. En effet, un moteur de DSE aura du mal à raisonner sur la longueur de `argv` et de déduire qu'une plus grande chaîne symbolique est nécessaire pour résoudre un problème. Très peu d'outils, dont `angr`, sont capables de faire varier la longueur de `argv` lors de l'exécution.

## 6.7 Couverture des chemins

### 6.7.1 Introduction

La couverture des chemins est une problématique commune à tous les types d'analyses *path-based* où le nombre croît approximativement quadratiquement. De plus, les programmes peuvent contenir des cycles potentiellement infinis. Il convient donc de prioriser les chemins les plus prometteurs pour les énumérer adéquatement.

**RQ 3.** *Comment traiter l'explosion combinatoire des chemins (induite par les boucles) ? En corolaire, comment explorer un programme de manière aussi exhaustive que possible en un temps raisonnable ?*

La couverture est dite *correcte* si tous les chemins sont exécutables avec les entrées générées. Autrement dit, chaque chemin est correct. La couverture est *complète* s'il n'existe pas d'entrées permettant de couvrir un nouveau chemin. D'un point de vue théorique, une exécution symbolique exhaustive fournit une méthode *correcte complète* pour n'importe quelle analyse décidable (*ce qui est rarement le cas d'un programme*). En pratique, le nombre de chemins d'un programme n'est jamais connu, ni même comment en couvrir un en particulier. (Une approche d'exécution symbolique probabiliste [134] a tenté de fournir

une réponse partielle en permettant d'estimer la probabilité de couvrir une portion d'un programme ou un chemin particulier). Néanmoins le problème de couverture reste entier, de par la combinatoire qu'il implique.

Pour obtenir une couverture du programme satisfaisante en limitant le problème d'explosion combinatoire des chemins, différentes mesures peuvent être prises. La première est l'élaboration d'une stratégie de couverture optimisée pour le critère de couverture voulu (cf. Section 6.7.2). Ensuite il est possible d'utiliser des approches d'exécution alternatives au DSE classique (cf. section 6.7.3). Enfin, plusieurs techniques permettent d'optimiser la recherche en réduisant la complexité du nombre de chemins, grâce à l'inférence, par exemple les invariants de boucle (cf. section 6.7.4).

## 6.7.2 Stratégies de couverture

Au-delà de la stratégie de couverture du programme, on discerne deux types d'évaluations de contraintes de chemin. La première, dite *eager evaluation*, résout un branchement conditionnel dès qu'il est rencontré avant d'aller plus loin. Cela correspond à une approche **Breath-First-Search (BFS)**. La seconde approche dite *lazy evaluation* résoud le prédictat de chemin que lorsque cela est nécessaire, ce qui correspond plus à une approche **Depth-First-Search (DFS)**. La première approche est privilégiée par la plupart des outils bien qu'elle requiert de conserver plus d'états.

**Stratégie orientée couverture** Les stratégies de couverture sont nombreuses, et à l'inverse du fuzzing, le DSE contrôle la manière dont est couvert le programme. Cela va de la stratégie classique comme **BFS**, **DFS** dans **DART** [137] aux stratégies à base de recherche générationnelle comme dans **SAGE** [139] et traités dans le chapitre 12 combinaisons d'analyses. La littérature est très active sur le sujet avec notamment en 2018 différents travaux d'optimisation de la recherche dans le programme [65, 357]. Le **BFS** est notamment préféré, car les prédictats de chemin sont nécessairement plus simples à résoudre et permettent d'explorer rapidement des chemins avec des entrées relativement robustes à la décohérence. **KLEE** [56] implémente plusieurs stratégies dont **random-path**. Il implémente aussi **cov-new**, une stratégie de probabilité sur les chemins fondée sur leur longueur, leur arité, le nombre de fois qu'ils ont été explorés et leur distance aux instructions non couvertes les plus proches. Il en choisit un au hasard pour continuer l'exécution, sachant que les chemins peu explorés sont favorisés ce qui limite l'épuisement de chemins créés par les boucles. Sur le bloc d'union des deux branchements d'un if, si les deux branches sont faisables, **KLEE** sélectionne systématiquement le chemin de "branchement" (if vrai) pour continuer l'exploration. Cela empêche parfois l'exploration de code. Ainsi la référence [127] propose la stratégie **random-state post-fork** (rspf) choisissant aléatoirement l'un des deux chemins. La référence [234] propose le **subpath-guided-search** qui favorise les sous-chemins peu explorés grâce à une structure stockant une distribution de la fréquence des sous-chemins explorés. Un chemin donné est subdivisé en sous-chemins de taille  $n$ . Cette taille joue un rôle essentiel dans les résultats, mais aucune valeur universelle n'est optimale pour tous les cas de test.

**Stratégies dirigées** Des approches plus dirigées ne visent pas une couverture maximale, mais plutôt d’atteindre des emplacements précis dans un programme. Les deux méthodes de base est d’utiliser la distance de fonctions sur le *call-graph* (Call Graph (CG)) ou bien la distance de basic-block sur le Control Flow Graph (CFG). Les travaux de Hicks [243] proposent une solution se basant sur un algorithme de plus court chemin dans le CFG interprocédural appelé *shortest-distance symbolic execution* (SDSE) mèle plus courte distance en basic-block dans une fonction le tout guidé par une plus courte distance entre les fonctions sur le CG. Les chemins ayant la plus courte distance à la l’emplacement cible sont favorisés de cette manière.

**Stratégie d’exploitation** Dans un contexte d’AEG, des stratégies visant à favoriser des chemins ayant des caractéristiques favorables à certains types de bugs ont été proposées [15]. L’une d’elles, le ***buggy-path first***, favorise les chemins ayant eu de petits bugs non exploitables (ex : off-by-one etc). L’intuition est qu’un chemin contenant ce genre d’erreurs à probablement moins été testé qu’un autre et contient donc potentiellement d’autres bugs eux exploitables. Une autre stratégie, appelée ***loop exhaustion***, explore les chemins qui itèrent les boucles en se basant sur l’observation qu’une mauvaise gestion des boucles favorise vraisemblablement les vulnérabilités de type buffer-overflow ou corruption mémoire. De manière analogue, Mayhem [64] favorise les chemins où sont effectuées des lectures/écritures symboliques. Une dernière optimisation effectuée par AEG, s’appelant *precondition symbolic execution*, consiste à contraindre les entrées du programme pour qu’ils satisfassent un ensemble de préconditions (*généralement orientées exploitation*). Cela permet de réduire drastiquement le nombre de chemins à couvrir. Ces préconditions sont généralement une taille connue pour un buffer, un préfixe dans une chaîne de caractères ou encore le contenu entier d’un buffer qui est déjà connu (dans un fichier par exemple).

**Autres stratégies** Certaines méthodes traditionnellement utilisées en fuzzing ont été appliquées au DSE comme l’ordonnancement de chemins à base de ***fitness function*** [366]. Ce mélange de stratégies entre techniques d’analyse est examiné dans le chapitre 12 combinaisons. Une dernière approche originale est celle de recherche guidée par des propriétés ***property-guided*** [379, 393]. L’idée est de représenter la propriété à vérifier sous forme de Finite State Machine (FSM) et ensuite guider le DSE en choisissant les chemins qui ont le plus de chances d’effectuer une transition sur la **FSM**.

### 6.7.3 Calcul en arrière

L’exécution symbolique arrière appelée Symbolic Backward Execution (SBE) [106] ou Backward-Bounded DSE (BB-DSE) [22], lorsque la recherche est bornée, est une variante du DSE classique (*en avant*). Cette méthode se rapproche du WP qui lui aussi fonctionne en arrière, mais sur une logique de Hoare. L’algorithme SBE part d’un point dans le programme et d’une propriété à prouver, et itère les chemins en arrière. À ce titre, cette

approche est dite *goal-oriented*. En principe, l'avantage est de pouvoir trouver une entrée permettant d'activer l'emplacement (*location*) dans le programme. En pratique, même en arrière, l'explosion combinatoire des chemins pose les mêmes problèmes qu'un SE en avant. De surcroît, les suffixes de chemins calculés ne sont pas nécessairement faisables en pratique. L'avantage d'effectuer la recherche en arrière et qu'un suffixe de chemin prouvé UNSAT permet d'éliminer tous les chemins ayant ce suffixe et donc d'élaguer des morceaux significatifs de l'arbre de recherche. Le SBE est donc particulièrement adapté pour résoudre des problèmes *d'infaisabilité* [22].

En termes de stratégie de couverture, les travaux [243] proposent la stratégie *call-chain backward symbolic execution* (CCBSE) dont le principe est de remonter les chemins fonction par fonction. Une fois l'entrée de la fonction atteinte avec un chemin, la stratégie réitère le processus avec les fonctions appelantes. Cela permet de segmenter la remontée en arrière en l'effectuant uniquement à cette granularité. Pour résumer, le CCBSE suit la chaîne d'appel en arrière depuis la cible, et au sein de chaque fonction effectue l'exploration de manière traditionnelle.

Des travaux sur une exécution dite “*symcretic*” [107] mélange approche avant et arrière dans une exploration en deux phases. La première fait du SBE jusqu'au point d'entrée du programme en marquant les contraintes problématiques. La deuxième phase s'effectue lorsqu'un point d'entrée est trouvé. Cette phase évalue concrètement le chemin en essayant de résoudre chaque contrainte marquée problématique via différentes heuristiques de recherche. De par l'exécution arrière, cette approche évite l'exploration en avant de chemins infaisables.

Pour conclure, le DSE en avant résout naturellement des problèmes d'atteignabilité *reachability* (faisabilité), tandis que le SBE/BB-DSE résout des requêtes d'infaisabilité. À moins d'atteindre le début du programme, une réponse SAT sur SBE ne permet pas de statuer sur la faisabilité du chemin. Néanmoins **éliminer les chemins infaisables réduit le problème d'explosion combinatoire**.

#### 6.7.4 Summarisation

La *summarisation* vise à approximer le comportement d'une fonction ou d'une boucle en modélisant logiquement les effets de bords. Cela soulève toutefois de nombreuses problématiques de recherche.

**Fonctions.** Une fonction pouvant être appelée plusieurs fois dans un même chemin il est intéressant de créer une seule fois un résumé synthétique logique de la fonction et de réutiliser les résultats précédemment calculés. Les travaux d'exécution symbolique *compositionnelle* [funsummary1] synthétisent les effets de bords à partir des entrées/sorties observées pour cette fonction lors de l'exécution concrète (cf. Section 12.5). Le résumé obtenu est une disjonction de formules de chemins intra procéduraux. D'autres travaux [7] étendent l'exécution symbolique compositionnelle en synthétisant les fonctions via des

formules logiques de premier ordre avec des fonctions non interprétées (*un-interpreted function*) qui permettent de modéliser partiellement une fonction. Enfin, [43] via des résumés de fonction réduit l'espace de recherche en éliminant des chemins produisant les mêmes effets de bords que de précédents chemins explorés.

**Boucles.** La création de résumés de boucles, *loop summarization*, nécessite d'inférer les (pre-)post-conditions de la boucle, mais aussi l'invariant de boucle lors de l'exécution symbolique [140]. Ce type de résumé évite au moteur d'exécution symbolique de dérouler la boucle. L'inconvénient est qu'il est très difficile d'inférer automatiquement cela et rapproche ce problème de ceux rencontrés par WP. De plus, les contraintes sont fortes : les boucles ne peuvent pas être imbriquées et ne peuvent pas contenir des branchements (conditions) à l'intérieur (*multi-path loops*). Ce dernier point semble être pris en charge par Proteus [367] qui effectue une analyse à base d'automate de dépendance de chemins *Path Dependency Automaton* (PDA) pour calculer les dépendances de chaque chemin aux entrées de la boucle. La formule finale générée est encore une disjonction des chemins possibles.

### 6.7.5 Fusion d'états

Aussi appelée *state-merging*, cette technique vise à fusionner différents chemins en un seul et même état à un emplacement commun du programme [164]. Intuitivement, la formule générée est la disjonction des deux chemins fusionnés. À l'inverse de l'interprétation abstraite, la fusion n'implique aucune approximation. Nonobstant le nombre réduit d'états à explorer, l'ajout de disjonctions dans les formules complexifie la résolution pour les solveurs. La question de recherche soulevée dès lors soulevée est :

**RQ 4.** *Quand et sous quelles conditions faut-il fusionner deux chemins en un seul état ?*

Les travaux de l'EPFL [217] sur cette question proposent deux approches. La première, le *query count estimation*, est une heuristique calculée statiquement estimant l'impact de chaque variable symbolique sur la difficulté d'une formule après un point de fusion. La deuxième méthode est une fusion dynamique de chemins effectuée de manière opportuniste lors de la sélection de chemins. La différence est que le point de fusion doit être un emplacement (adresse) commun aux deux états alors que la fusion dynamique ne requiert pas nécessairement de point de jointure<sup>54</sup>. Par ailleurs, la fusion statique est plus adaptée pour une couverture complète du programme alors que la fusion dynamique permet des stratégies plus dirigées.

Le *Veritesting* [16] est une autre approche de fusion de chemin statique. Il s'appuie sur différentes heuristiques basées sur des instructions dites “simples” et “dures”. Les instructions “dures” sont notamment les sauts indirects, les appels système ou les instructions qui

---

54. <https://blog.trailofbits.com/2019/01/25/symbolic-path-merging-in-manticore/>

ne sont pas modélisées de façon exacte. Les séquences d'instructions simples sont fusionnées avec l'expression SMT `ite`, tandis qu'une exécution symbolique standard “*per-path*” est toujours effectuée pour les instructions dures.

## 6.8 Résolution des formules

### 6.8.1 Introduction

La résolution de formules à base de contraintes est la clé de voûte du DSE et de nombreuses autres analyses de programmes appliquées à la sécurité [324]. Un solveur est une procédure de décision pour des problèmes exprimés sous forme de formules logiques. De nombreux problèmes peuvent par exemple être exprimés sous la forme de problèmes de *satisfaisabilité* booléenne, couramment appelés SAT, dont le but est de trouver une valeur pour chaque booléen rendant le formule vraie. Bien que ce problème soit NP-complet, les solveurs SAT ont progressé de manière fulgurante, notamment grâce à leur utilisation pratique au sein des solveurs SMT [256]. Les solveurs SMT généralisent le problème SAT dans lequel les variables booléennes sont remplacées par des prédictats dans différentes théories, permettant d'exprimer des problèmes de manière plus naturelle.

#### Attention

On utilise souvent à tort le terme “solveur de contraintes” pour désigner un solveur SMT. Or, à proprement parler, un solveur de contraintes issu du [Constraint Programming \(CP\)](#) fonctionne différemment d'un solveur SMT. L'annexe D décrit la différence entre les deux.

Bien que les solveurs aient connu une évolution conséquente grâce aux applications liées à la vérification logicielle, ils restent malgré tout la principale entrave au passage à l'échelle du DSE. Cela pose la question de recherche suivante :

**RQ 5.** *Comment optimiser les formules et limiter les traitements non linéaires qui se retrouvent inéluctablement dans les formules logiques générées ?*

Les solveurs s'améliorent d'année en année notamment grâce à la compétition SMT qui oppose les solveurs dans les différentes catégories de théories<sup>55</sup>. Ces améliorations sont strictement internes aux solveurs et indépendantes d'un usage dans un contexte d'exécution symbolique. Toutefois, différentes solutions sont proposées au niveau DSE pour améliorer le passage à l'échelle de la résolution de formules (cf. section 6.8.5).

---

55. <https://smt-comp.github.io/>

### Note

Les travaux [363] proposent une approche originale visant choisir le solveur à utiliser avec du machine learning. L'idée est d'entrainer le classifieur avec différentes formules pour apprendre quel solveur est le plus adapté en fonction du contenu et des théories utilisées. Lors de l'exécution, le choix est alors effectué automatiquement par le modèle entrainé. Les auteurs annoncent un gain en temps de résolution variant de 10 à 42%.

## 6.8.2 Mode interactif

La plupart des solveurs intègrent maintenant un mode interactif permettant de recevoir dynamiquement les clauses et les commandes SMT directement sur `stdin` et non plus par fichier. Couplé avec les commandes `push` et `pop`, il est possible d'ajouter et de retirer des clauses sans que le solveur ait à réévaluer toutes les clauses précédentes. Utilisé à bon escient, cela permet de résoudre plusieurs prédictats de chemins dans une seule instance du solveur.

## 6.8.3 Théories

Il existe un certain nombre de théories en SMT dont les interfaces et les opérations sont standardisées dans le format/langage SMT-LIB 2.6 [23] maintenant largement adopté par les solveurs. Les théories spécifiées sont : entiers, tableaux, bitvecteurs (de taille fixe), core (booléens), nombres flottants, réels et entiers et réels (entremêlés). Certains analyseurs utilisent la théorie des entiers pour représenter les valeurs entières. Néanmoins, la théorie considère les entiers comme mathématiques donc infinis. Les deux théories généralement considérées sont bitvecteurs et tableaux qui permettent de modéliser toutes les opérations d'un programme (*à l'exception des flottants rarement modélisés*). À titre d'exemple, la figure 6.3 présente la théorie des tableaux qui se définit avec deux opérations `Sel` et `store`. Le type `Array  $\mathcal{I}$   $\mathcal{E}$`  désigne un tableau paramétré par deux types  $\mathcal{I}$  et  $\mathcal{E}$  qui sont respectivement le type des index et le type des valeurs du tableau. La colonne de droite donne les propriétés inhérentes aux deux opérations.

$$\begin{array}{ll} \text{select} : \text{Array } \mathcal{I} \mathcal{E} \times \mathcal{I} \rightarrow \mathcal{E} & \forall a i e. \text{select}(\text{store } a i e) i = e \\ \text{store} : \text{Array } \mathcal{I} \mathcal{E} \times \mathcal{I} \times \mathcal{E} \rightarrow \text{Array } \mathcal{I} \mathcal{E} & \forall a i j e. (i \neq j) \Rightarrow \text{select}(\text{store } a i e) j = \text{select } a j \end{array}$$

FIGURE 6.3 – Théorie des tableaux

Plusieurs théories se combinent au sein d'une même formule sous la forme de “logique” dont le nom suit une nomenclature précise. Ainsi, QF\_ABV désigne une formule sans quantificateurs (QF *Quantifier Free*) pour A *array* et BV les bitvecteurs. La plupart

des logiques peuvent être exprimées avec ou sans les opérateurs de quantification ( $\forall$ ,  $\exists$ ) pour créer des formules d'ordre supérieur. Celles-ci sont évidemment beaucoup plus dures à résoudre. Parmi les autres logiques, il y a IA *Integer Artithmetics* préfixé ou non de *Linear* ou de *NON-LINEAR* etc.

#### Théorie des strings

La théorie des strings est une théorie non standardisée dans SMTLIB qui commence à émerger dû à la difficulté de résoudre des contraintes sur les chaînes de caractère avec les primitives sur les arrays. Cette théorie fournit notamment les opérations `Concat`, `Length` ou `Contains`. Les deux dernières sont des métapropriétés associées aux chaînes de caractères sur lesquelles les DSE standards ont du mal à raisonner (cf. 6.9). Cette théorie fournit aussi des opérations liées aux expressions régulières. L'une des premières implémentations fut `Z3-str` dans `Z3` [395] ou encore l'implémentation comme `Trau` [4] qui surpassé l'état de l'art en termes de performances.

### 6.8.4 Outils de résolution de contraintes

Le plus connu des solveurs est sans conteste `Z3` [257] développé par Microsoft et utilisé pour un très grand nombre d'applications différentes. Ses différents bindings le rendent très facile d'utilisation au sein d'un moteur d'exécution symbolique et il est par exemple utilisé dans `Mayhem` [64], `SAGE` [139], `angr` [320] ou encore `Triton` [304]. Il inclut le plus grand nombre de théories parmi lesquelles bitvecteurs, tableaux, quantificateurs, fonctions non interprétées, entiers (linéaires et non linéaires), réels, et récemment une théorie expérimentale sur les strings avec `Z3-str` [395]. Ensuite vient `STP` [129, 130], utilisé dans `EXE` [57], `KLEE` [56] ou encore `AEG` [15].

Le tableau 6.2 ci-dessous résume les principaux solveurs SMT, leur solveur SAT sous-jacent, ainsi que leurs différentes caractéristiques. Les solveurs ne supportant ni les bivecteurs ni les tableaux sont généralement spécifiques à des logiques particulières (`SMT-RAT` pour `NRIA` ou `veriT` pour toutes les logiques sur les entiers). Les deux dernières colonnes indiquent les résultats des solveurs à la compétition de SMT organisée annuellement dans les deux logiques cibles de cet état de l'art `QF_BV`<sup>56</sup> et `QF_ABV`<sup>57</sup>. `Boolector` tient la tête du podium dans les deux catégories en 2018, et cela était déjà le cas depuis plusieurs années consécutives.

Le standard SMT-LIB fourni déjà une interface commune de formules entre solveurs que certains outils comme `metaSMT` [296] utilisent pour fournir un frontend commun pour différents solveurs. Cela permet à un outil utilisant `metaSMT` d'interchanger le solveur facilement. Historiquement `KLEE` fournit un binding direct pour `STP` et `Z3`, mais il possède aussi maintenant un binding vers `metaSMT`<sup>59</sup>. `angr` utilise un module Python appelé

56. [http://smtcomp.sourceforge.net/2018/results-QF\\_BV.shtml](http://smtcomp.sourceforge.net/2018/results-QF_BV.shtml)

57. [http://smtcomp.sourceforge.net/2018/results-QF\\_ABV.shtml](http://smtcomp.sourceforge.net/2018/results-QF_ABV.shtml)

58. mais libre pour une utilisation non commerciale

59. <https://klee.github.io/docs/solver-chain/>

	OSS	Théorie		Inc	API	SAT	SMTCOMP 2018	
		BV	A				QF_BV	QF_ABV
Alt-Ergo [183]	✓	✓	✓	n/c	Ocaml	-	-	-
Boolector [50]	✓	✓	✓	n/c	C, Py	Lingeling, CaDiCaL PicoSAT, MiniSAT	1 <sup>er</sup>	1 <sup>er</sup>
CVC4 [102]	✓	✓	✓	n/c	C++, Java	-	3 <sup>ème</sup>	4 <sup>ème</sup>
MathSAT5 [82]	✓	✓	✓	n/c	C, Py	-	-	-
OpenSMT2 [11]	✓	✓	✓	n/c	n/c	MiniSAT	-	-
raSAT [205]	✓	✗	✗	n/c	n/c	-	-	-
SMTInterpol [80]	✓	✗	✓	n/c	n/c	n/c	-	-
SMT-RAT [90]	✓	✗	✗	n/c	C++	-	-	-
STP [129]	✓	✓	✓	n/c	C, Py	[Crypto]MiniSAT	4 <sup>ème</sup>	-
veriT [45]	✓	✗	✗	n/c	n/c	-	-	-
Yices2 [114]	✓	✓	✓	n/c	C	n/c	-	2 <sup>ème</sup>
Z3 [257]	✓	✓	✓	✓	C++, Py	n/c	-	3 <sup>ème</sup>
Minkeyrink [163]	✗ <sup>58</sup>	✓	✓	n/c	n/c	n/c	2 <sup>ème</sup>	-
ABC [47]	✓	✓	✗	n/c	✗	Glucose	-	-

TABLE 6.2 – Tableau comparatif de plusieurs solveurs SMT existants

claripy<sup>60</sup> effectuant approximativement la même tâche.

### 6.8.5 Optimisation de formules

Au niveau exécuteur symbolique, les différentes optimisations et améliorations de gestion de formules proposées dans la littérature sont les suivantes :

**Pré traitement de la formule** : Bien que le solveur effectue de nombreuses passes de simplification, les exécuteurs symboliques appliquent souvent des passes de propagation de constantes ou de réécriture d'expressions pour obtenir une formule plus canonique. S2E [77] simplifie par exemple les expressions de masque sur les bitfields en remplaçant les bits ignorés par le masque par une constante.

**Réutilisation de solutions** : Pour éviter de relancer une résolution de formule, il faut être en mesure d'évaluer l'équivalence sémantique, ou dans une moindre mesure l'équivalence syntaxique. EXE [57] dispose d'un cache des résultats de formules partagé entre différentes instances explorant différents chemins. À l'inverse, KLEE [56] conserve un cache de contre-exemples. Ensuite, si une formule contient une sous-formule sauvegardée comme UNSAT, alors la formule est considérée UNSAT. Inversement, si une formule est une sous-formule d'une formule sauvegardée comme SAT, alors celle-ci est aussi SAT. D'autres approches [351, 375] proposent notamment des techniques de mémoïsation et de réutilisation de sous-formules

60. <https://docs.angr.io/advanced-topics/claripy>

**formules sous-constraints** : Cette optimisation se base sur le fait qu'un temps non négligeable est passé à résoudre des divisions et en particulier des restes avec des dénominateurs symboliques. Lors de l'exécution symbolique d'une opération coûteuse, le DSE va ajouter une contrainte "fainéante" sur le résultat de l'opération pour les deux branches [289]. Cette approche se combine avec une évaluation *lazy* des chemins qui permet de retarder la résolution d'un chemin pouvant contenir des contraintes non linéaires.

**Optimisations sur les tableaux.** Sur les formules, une attention particulière est donnée pour optimiser les opérations sur les tableaux [119, 278] car celles-ci sont très coûteuses pour les solveurs. Une optimisation consiste à simplifier les lectures et les écritures en appliquant deux règles, le read-over-write, noté RoW, et le write-over-write, noté WoW [119]. Le RoW vise instinctivement à vérifier si une valeur lue a précédemment été écrite. Il se base sur la propriété de *fonctionnalité* qui stipule que pour un tableau  $A$  et deux index  $i, j$  égaux, un **select** sur l'un ou l'autre des index sont égaux. De surcroît, le RoW et le WoW considèrent les axiomes donnés en figure 6.4 :

$$\textbf{RoW1} : \text{si } i = j, \text{select}(\text{store}(a, i, v), j) = v \quad (6.1)$$

$$\textbf{RoW2} : \text{si } i \neq j, \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j) \quad (6.2)$$

$$\textbf{WoW1} : \text{si } i = j, \text{store}(\text{store}(A_n, i, v_1), j, v_2) \implies \text{store}(A_n, j, v_2) \quad (6.3)$$

FIGURE 6.4 – Axiomes sur RoW et WoW

L'axiome **RoW1** implique qu'on peut remplacer le **select** par la valeur écrite précédemment si les index (logiques) sont les mêmes. **RoW2** indique que si deux index logiques sont différents, il est possible d'effectuer le **select** sur le tableau  $a$  et non pas sur le tableau résultant de l'écriture. Enfin, **WoW1** implique que si une valeur est écrasée sans qu'elle ait été lue par ailleurs, il est possible d'ignorer la première écriture. La mémoire n'est qu'une chaîne d'écritures successives. Il est donc possible de remonter cette chaîne pour appliquer les règles **RoW1**, **RoW2** et **WoW1** pour chaque lecture/écriture symbolique effectuée sur la mémoire. Cela pose le problème d'une complexité qui peut devenir quadratique en la taille de la chaîne de **store**. Des structures de données permettant d'obtenir un bon compromis entre simplification et complexité ont donc été élaborées [97, 119]. Cette optimisation s'applique sur toute formule logique SMT qu'elle soit issue du DSE sur un programme ou toute autre application.

En sus, si tous les index sont constants dans la formule, il est possible de transformer une formule QF\_ABV en QF\_BV via une opération de *memory flattening* logique [97].

## 6.9 Analyse semi-automatisée, annotations manuelles

Toutes les problématiques évoquées dans les sections précédentes expliquent pourquoi le passage à l'échelle du DSE est très délicat dans un contexte d'analyse automatisée. Très peu de travaux visent à évaluer l'apport d'annotations manuelles pour guider et aider la recherche de bugs. C'est précisément ce qu'ont fait John Galea et Sean Heelan [127] appliqué à KLEE. Ces travaux étudient les constructions de code et les motifs sur lesquels un moteur de DSE (KLEE) va s'enliser. Voici les problèmes empiriques qui requièrent généralement une assistance manuelle pour être surmontés.

**Explosion combinatoire de chemins.** L'un des problèmes est que le DSE va perdre du temps et fork des états sur du code inutile (fonctions de logs etc). Ceci est légèrement limité par `uClIBC`<sup>61</sup> qui fournit des stubs pour certaines fonctions (à moins que les arguments soient symboliques). Une solution est donc de *systématiser l'utilisation de stub symbolique de faible fidélité par rapport au comportement réel de la fonction*. Un autre problème est le fork d'états sur des fonctions vérifiant un délimiteur ou un caractère de terminaison. Cela implique un déroulement de boucle inutile alors qu'il suffit d'ajouter un `klee_assume` indiquant que le caractère cherché est bien trouvé.

**Chemins d'erreurs.** Un constat effectué est que le DSE perd du temps et des ressources à essayer de couvrir des chemins d'erreur qui ne retournent jamais au point initial du CFG (ex : code de retour sur la plupart des fonctions de la libc). Une solution invasive pour résoudre ce problème consiste à mettre un `exit` au plus haut dans la fonction de gestion d'erreur. Cela élague intrinsèquement le nombre de chemins à couvrir.

**Initialisation de données statiques.** Beaucoup de programmes commencent leur exécution en initialisant des données utilisées (ou non) dans la suite de l'exécution. Cela implique une surcharge inutile dans le prédictat de chemin (e.g. `libjpeg` initialize des tableaux statiques au démarrage qui totalisent 19.8 millions d'instructions LLVM). Une solution invasive consiste à précalculer ces structures et à les intégrer dans le programme statiquement.

**Incapacité de raisonner sur les métapropriétés.** Le DSE peut difficilement, d'un point de vue logique, établir la relation entre une chaîne symbolique et sa longueur qui ne sera pas symbolique, bien qu'intégralement calculée sur des données symboliques. De la même manière, un code qui comptera le nombre d'occurrences d'un caractère crée une métapropriété sur la valeur qui ne sera pas symbolique. Une solution est de symboliser la longueur sur l'entièreté de la chaîne symbolique.

Les solutions à ces différents problèmes peuvent difficilement s'automatiser et requièrent une intervention humaine et une compréhension du code pour être effectives.

---

61. <https://github.com/klee/klee-uclIBC>

Dans tous les cas, il est toujours profitable que le DSE fournisse des mécanismes d'annotation permettant d'influencer sur les comportements de l'algorithme de SE et sur la couverture.

#### Annotations dans Triton

Des expérimentations ont été faites avec **Triton** pour supporter un mécanisme identique à KLEE avec `triton_assume` et `triton_assert`, permettant respectivement de considérer l'expression donnée en paramètre comme vraie ou tester la validité de celle-ci. Il suffit ensuite de compiler le programme en le linkant sur une bibliothèque contenant un *stub* pour ces deux fonctions. Lors du DSE, il est alors facile de repérer l'appel à ces fonctions et les traiter en conséquence.

## 6.10 Outils d'exécution Symbolique

En termes d'outils, n'ont été retenus que les outils permettant de tester des programmes C ou C++ sous la forme source ou binaire. Ainsi, tous les moteurs d'exécution symbolique sur le Java, Javascript et autres, ont intentionnellement été ignorés. Les différentes ressources de l'état de l'art ont permis d'identifier les outils référencés dans le tableau 6.3. Les moteurs de DSE couplés à du fuzzing ne figurent pas dans le tableau et sont référencés dans la partie Combinaison (cf. chapitre 12). Le tableau reflète l'état de l'art des outils (*début 2019*) et certaines informations sont potentiellement parcellaires par manque d'information ou de documentation concernant l'outil.

Les critères de comparaison regroupent certaines caractéristiques discriminatoires entre les outils. Parmi celles-ci la gestion de la stratégie de couverture, la direction de l'analyse (avant/arrière) ou encore la gestion des lectures/écritures mémoires. Pour ce dernier  $\mathcal{S}$  indique que le moteur supporte les index symboliques et donc aussi les index concrets (l'inverse n'est pas vrai pour  $\mathcal{C}$ ). Ce tableau est rempli de manière opportuniste en raison de la difficulté d'obtenir toutes les informations concernant chaque colonne pour chaque outils. Il est possible que certaines cellules n'aient pas été remplie alors que l'outil satisfait la caractéristique. Ce tableau ne reflète pas le caractère maintenu des outils qui pour certains sont largement abandonnés. Le chapitre 7 détaille les caractéristiques des outils sélectionnés et des ressources étudiées.

## 6.11 État de l'art : Conclusion intermédiaire

Les sections précédentes ont survolé les différents aspects du DSE, qui se révèle être une technique potentiellement très efficace, mais très difficile à maîtriser pour en obtenir des résultats. À l'inverse des fuzzers étudiés dans le chapitre suivant (cf. 9) cette technique requiert une connaissance de la sémantique du programme. Cela donne un pouvoir

	Caractéristiques				Exec.	Couv. chemins	Contraintes
	Open-source	Source	Type				
AEG [15]	✓		Binaire				
angr [320]	✓	✓	✓	✓			
BAP [49]	✓	✓	✓	✓			
Binsec [99]	✓	✓	✓	✓			
BitBlaze [326]	✓	✓	✓	✓			
Bouncer [91]	✓	✓	✓	✓			
Cloud9 [52]	✓	✓	✓	✓			
CREST [53]	✓	✓	✓	✓			
CUTE [311]	✓	✓	✓	✓			
DART [137]		✓	✓	✓			
EXE [57]		✓	✓	✓			
Fuzzball [248]	✓	✓	✓	✓			
KLEE [56]	✓	✓	✓*	✓			
Miasm [255]	✓	✓	✓	✓			
Mayhem [64]		✓	✓	✓			
Manticore [33]	✓	✓	✓	✓			
McVeto [341]		✓	✓	✓			
Otter [292]	✓	✓	✓	✓			
Pathcrawler [364]	✓	✓	✓	✓			
Pathgrind [315]	✓	✓	✓	✓			
Reven [340]		✓	✓	✓			
SAGE [139]		✓	✓	✓			
S2E [77]	✓	✓	✓	✓			
Triton [304]	✓	✓	✓	✓			

\* : grâce aux “lifter” depuis le binaire (cf. 6.3.3)

TABLE 6.3 – Comparaison des caractéristiques des outils d'exécution symboliques

	<b>un chemin <math>\pi</math></b>	<b>couverture des chemins <math>\Pi^P</math></b>
<b>correct</b>	$\varphi_\pi$ est satisfiable et permet de générer des entrées couvrant le chemin	tous les chemins sont satisfiables et peuvent être exercés
<b>incorrect</b>	les entrées ne permettent pas forcément de couvrir le chemin	certaines chemins sont potentiellement infaisable
<b>complet</b>	toutes les entrées exerçant $\pi$ sont des solutions à $\varphi_\pi$	il n'existe pas de chemin exerceable qui ne soit pas dans $\Pi^P$ (impossible en pratique)
<b>incomplet</b>	le chemin est exerceables par des entrées qui rendent $\varphi_\pi$ insatisfiable	certain chemins exerceables n'ont pas été couverts

TABLE 6.4 – Résumé correction, complétude DSE

de raisonnement très puissant à l'analyse, mais cela peut en devenir le principal défaut si la sémantique est mal modélisée. Le DSE se résume en une gestion contrôlée des approximations qui sont faites sur la sémantique du programme. Les deux leviers étant les concrétiisations et les symbolisations, il convient de les alterner avec parcimonie. Le premier permettant de passer à l'échelle tout en réduisant la complétude et le deuxième permettant d'être complet au détriment de la correction. Le tableau 6.4 résume l'implémentation de la correction sur un chemin ou la couverture.

En termes de recherche, le DSE pose de nombreuses questions évoquées dans les sections précédentes et qui ne sont toujours pas résolues à ce jour. Le tableau 6.5 résume les différentes questions de recherches soulevées par ce domaine.

<b>Id</b>	<b>Description</b>
RQ_DSE1	Comment traiter la mémoire, l'aliasing de pointeurs et les lectures écritures à des adresses symboliques ? Une question subséquente est comment modéliser la segmentation mémoire ?
RQ_DSE2	Comment traiter les interactions avec l'environnement ? Autrement dit comment modéliser symboliquement les effets de bord de ceux-ci en minimisant la décohérence ?
RQ_DSE3	Comment traiter et limiter l'explosion combinatoire des chemins ( <i>induites par les boucles</i> ) ? Le corollaire est comment guider la couverture efficacement pour maximiser celle-ci ?
RQ_DSE4	Quand et sous quelles conditions est-il profitable de fusionner deux chemins en un seul état ?
RQ_DSE5	Comment optimiser les formules pour limiter les traitements non-linéaires qui se retrouvent inévitablement dans les formules de prédictat de chemin ?

TABLE 6.5 – Récapitulatif des questions de recherche en DSE

### *6.11. État de l'art : Conclusion intermédiaire*

---

Ces différentes questions de recherche soulèvent chacune différents problèmes que la littérature s'attèle à résoudre. RQ\_DSE3 est néanmoins la question de recherche la plus active, car c'est celle sur laquelle l'analyse a le plus de marge de manœuvre et le meilleur potentiel d'amélioration. En effet, RQ\_DSE2 requiert en majeure partie de gros efforts de développement et RQ\_DSE5 relève en partie du domaine de la logique et de la recherche en procédure de décision (*très distincte du domaine la sécurité*).



## Analyse détaillée des outils

---

### 7.1 Outils de DSE sélectionnés

Parmi les différents outils mentionnés dans le tableau 6.3, `Mayhem` [64], `angr` [angr], `KLEE` [56], `Manticore` [33] et `Triton` [304] ont été retenus. `Mayhem`, bien que non disponible, fut l'un des pionniers et fait référence dans le domaine. Dans une moindre mesure, la connaissance profonde de `Triton` par Quarkslab rendait son analyse aisée dans le cadre de l'état de l'art, celui-ci étant activement développé et maintenu par Quarkslab. Enfin, `KLEE` et `angr` sont aussi des références incontournables en DSE et `Manticore` semble très prometteur notamment pour son support ARM et sa facilité d'utilisation. Enfin, `S2E` bien qu'abondamment cité, a été écarté des tests, car il se base essentiellement sur `KLEE`. Sa plus-value se trouve dans son aptitude à “lifter” du binaire vers du `LLVM`, or, dans le contexte de l'étude du projet le code est à disposition.

Le tableau 7.1 synthétise toutes les caractéristiques détaillées de chacun des outils, avec d'une part les critères d'évaluations des outils (`CTO`) communs à toutes les techniques d'analyse et d'autre part les `CTO` spécifiques à l'exécution concolique (noté `CTO_C`). Ce dernier est divisé en sous-catégories traitant les différents aspects du DSE, à savoir la gestion des entrées symboliques, la modélisation de la mémoire, le calcul d'un chemin, la couverture des chemins et la résolution de formules. Chacune de ces catégories contient les différents critères d'évaluation.

**Caractéristiques.** Ces critères définissent si l'exécution s'effectue sur une représentation du code source ou une représentation du binaire (`CTO_C1`), si l'exécution est purement symbolique ou si elle s'appuie sur des valeurs concrètes et une instrumentation (`CTO_C2`), ou encore la direction de l'analyse (`CTO_C5`). Le chargement du programme `CTO_C4` indique si toutes les instructions sont décodées (“liftées”) vers l'IR une seule fois, ou bien si une instruction est décodée à chaque fois qu'elle est rencontrée sur un chemin. `CTO_C3` et

---

62. via Symbion (cf. 7.5.7)

## Chapitre 7. Analyse détaillée des outils

			#1 Mayhem	#2 Manticore	#3 KLEE	#4 angr	#5 Triton
CT0	Général outils	CT0_1 : Langage	C++, Oc	Py	C++	Py	C++, Py
		x86	✓	✓	n/a	✓	✓
	CT0_2 : Architecture	x86-64	✓	✓	n/a	✓	✓
		ARMv7	✓	✓	n/a	✓	✗
		ARMv8	✗	✗	n/a	✓	✓
	CT0_3 : Open-source		✗	✓	✓	✓	✓
	CT0_4 : Licence	/	AGPL-3.0	NCSA	BSD-2	Apache 2.0	
CT0_C	Caractéristiques	CT0_5 : Documentation	/	✗	✓	✓✓	✓
		CT0_6 : Activité	✓✓	✓✓	✓	✓✓	✓✓
		CT0_7 : Jeu de tests	/	/	/	/	/
		CT0_C1 : Base	source		✓		
			binaire	✓	✓	✓	✓
		CT0_C2 : Type	symbolique		✓	✓	✓
			concolique	✓	✓	✓ <sup>62</sup>	✓
CT0_C	Entrées	CT0_C3 : Format trace	n/c	n/c	n/c	n/c	custom
		CT0_C4 : Chargement	programme		✓	✓	
			par inst.	✓	✓		✓
		CT0_C5 : Direction	en avant	✓	✓	✓	✓
			en arrière	✗	✗	✗	✗
		CT0_C6 : Représentation intermédiaire	BAP	∅	LLVM	VEX	custom
		CT0_C7 : argv variable	n/c	✗	✗	✓	n/a
CT0_C	Mémoire	CT0_C8 : Bibliothèques	n/c	~	✓	✓✓	n/a
		CT0_C9 : Appels systèmes	30	✓✓	✓	✓✓	n/a
		CT0_C10 : Instrs. Non déterministes	n/c	~	~	n/c	~
		CT0_C11 : Lecture symbolique		✓	✓	✓	✗
		CT0_C12 : Écriture symbolique	✗	✗	~	✓	✗
		CT0_C13 : Procédure décision		✓	✓	✓	n/a
		CT0_C14 : Type	state-forking	n/c	✓	✓	n/a
CT0_C	Calc. $\pi$		if-then-else	n/c	✓	✓	n/a
		CT0_C15 : Type	online	✓	✓	✓	✓
			offline	✓			✓
		CT0_C16 : Optimisations	slicing	✗	✗	✓	✓
			tainting	✓	✓	✓	✓
		CT0_C17 : Stratégie couverture		✓	✓	✓✓	✓
		CT0_C18 : Dirigé		✗	✗	n/c	n/a
CT0_C	Couverture II	CT0_C19 : Optimisations	path-merging	✓	~	✓	n/a
			infeas. $\pi$ pruning	n/c	✗	n/c	n/a
			fun. summary	n/c	✗	n/c	n/a
			loop summary	n/c	✗	n/c	n/a
		CT0_C20 : Type	solveur contrainte				
			solveur SMT	✓	✓	✓	✓
Résolution formules	Résolution formules	CT0_C21 : Solveurs supportés	Z3	Z3	*	Z3	Z3
		CT0_C22 : Théories	bit-vecteurs	✓	✓	✓	✓
			tableaux	??	✓	✓	✗
		CT0_C23 : Export SMT2		n/c	✗	✓	✓
		CT0_C24 : Optimisations	tableaux	✓	~	✓	n/a
			cache formules	✓	~	✓	~

TABLE 7.1 – Comparaison des outils sélectionnés

CTO\_C6 indiquent respectivement un éventuel format de trace utilisé et la représentation intermédiaire utilisée.

**Gestion des entrées.** CTO\_C7 indique si le moteur de DSE est capable de faire varier la taille des entrées symboliques données via `argv`. CTO\_C8, CTO\_C9 et CTO\_C10 montrent de manière plus quantitative le nombre d'appels système, d'appels de bibliothèques et le nombre d'instructions non déterministes modélisées symboliquement (*toutes plateformes confondues*).

**Modèle mémoire.** dénote l'aptitude du moteur de DSE à modéliser les accès mémoire et notamment les lectures, écritures symboliques (CTO\_C11, CTO\_C12), et si une quelconque procédure de décision permet de moduler le comportement pour chaque pointeur lu ou écrit (CTO\_C13). À ce titre, `Mayhem` pourrait supporter les écritures symboliques, mais par souci de performance ce n'est pas le cas. Il utilise par ailleurs une analyse par interprétation abstraite pour décider ou non de la concrétisation des pointeurs lus (cf. 6.4.2). CTO\_C14 définit le comportement adopté en cas de pointeur symbolique (voir Stratégies de couverture des valeurs de pointeurs 6.5.2).

**Calcul d'un chemin ( $\pi$ ).** définit les caractéristiques du traitement d'une trace, à savoir si elles sont indépendantes les unes des autres (offline), ou si pour une même exécution, plusieurs chemins sont gérés en parallèle (CTO\_C15). Enfin, CTO\_C16 indique si des analyses annexes de slicing ou tainting viennent se greffer concomitamment au calcul du prédictat de chemin.

**Calcul de la couverture des chemins ( $\Pi$ ).** CTO\_C17 indique si une stratégie de couverture intelligente est appliquée pour de la couverture ou de la recherche de vulnérabilités (*donc autre que du DFS, BFS classique*). `Mayhem`, par exemple, favorise les chemins comportant plus d'accès symboliques, car ceux-ci sont plus propices à erreur. CTO\_C18 indique si la stratégie de recherche est dirigée vers différents emplacements identifiés à l'avance par d'autres analyses *ad-hoc*. Enfin, CTO\_C19 indique les différentes optimisations qui peuvent être appliquées pour essayer de minimiser le nombre de chemins à traiter et pallier le problème de complétude de la couverture.

**Résolution de contraintes.** indique notamment le type de solveur (CTO\_C20), les solveurs supportés (CTO\_C21), ainsi que les différentes théories supportées (CTO\_C22). Enfin CTO\_C24 englobe toute optimisation qui pourrait être faite par le moteur de DSE.

Les sections ci-dessous élaborent sur les différents outils et leurs différentes caractéristiques présentées succinctement dans le tableau.

## 7.2 DSE #1 : Mayhem

Mayhem a fait autorité pendant plusieurs années en tant qu'exécuteur symbolique dynamique de binaires appliqué à la recherche de vulnérabilités (notamment grâce à ses stratégies innovantes de concrétisation de pointeurs). Il fut initié par des travaux de recherche mais il est désormais activement développé et maintenu par la startup ForAllSecure (ou VSecure) fondée par les mêmes chercheurs ayant initié le projet. Ce qui a entériné la décision d'examiner cet outil est son intronisation comme meilleur CRS lors du challenge CGC<sup>63</sup> avec sa 1<sup>ère</sup> place et un prix de 2 millions de dollars<sup>64</sup>. Le moteur de DSE n'est décrit que par une seule publication académique qui possède tout de même plus de 300 citations. La publication exacte est :

*“Unleashing Mayhem on Binary Code” par Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert et David Brumley publié au IEEE Symposium on Security and Privacy, S&P 2012*

Cet article possède l'ISBN : 978-0-7695-4681-0 et le DOI 10.1109/SP.2012.31. Tous les auteurs étaient alors affiliés à l'université Carnegie Mellon, Pittsburgh, PA.

### 7.2.1 Contributions

Les contributions du papier ciblent les deux plus grosses problématiques du DSE qui sont 1) gérer les chemins en mémoire (*pour éviter la pénurie de mémoire*) et 2) résonner sur les index symboliques en mémoire. Pour résoudre ces deux problèmes, ils proposent les deux solutions suivantes :

- un moteur d'exécution concolique *hybride* (cf. 7.2.4) alternant une exécution *online* et une exécution *offline* (cf. Section 6.4.1).
- un modèle mémoire basé sur les index (*index-based*) (cf. 7.2.5) permettant de résonner efficacement sur les pointeurs symboliques.

En outre, ils proposent une méthode pour détecter des vulnérabilités et surtout démontrer leur exploitabilité via des requêtes SMT additionnelles. Les résultats obtenus sont significatifs. Ils ont permis de démontrer l'exploitabilité de 29 bugs dont 2 étaient inconnus. Arriver à prouver l'exploitabilité d'un bug et non pas juste la présence du bug est un élément central de l'approche.

### 7.2.2 Considérations de conception

La conception de Mayhem s'articule autour des quatre points suivants :

---

63. <https://www.youtube.com/watch?v=n0kn4mDXY6I>

64. [https://techcrunch.com/2016/08/05/carnegie-mellons-mayhem-ai-takes-home-2-million/from-darpas-cyber-grand-challenge/](https://techcrunch.com/2016/08/05/carnegie-mellons-mayhem-ai-takes-home-2-million-from-darpas-cyber-grand-challenge/)

1. le moteur doit invariablement fonctionner sans épuiser les ressources du système
2. le moteur ne doit pas répéter du travail déjà fait
3. le moteur ne doit pas jeter des résultats qui pourraient être utilisés par des itérations ultérieures
4. le moteur doit raisonner sur une mémoire symbolique pour les lectures/écritures contrôlées par l'utilisateur

Pour satisfaire au mieux ces contraintes, le moteur s'articule autour de deux processus fonctionnant en parallèle. Le premier, le *Concrete Executor Client*, noté CEC, exécute le code nativement et applique la propagation de teinte dynamique. Le second, le *Symbolic Executor Server*, noté SES, effectue les tâches d'exécution symbolique et la gestion d'ordonnancement des chemins.

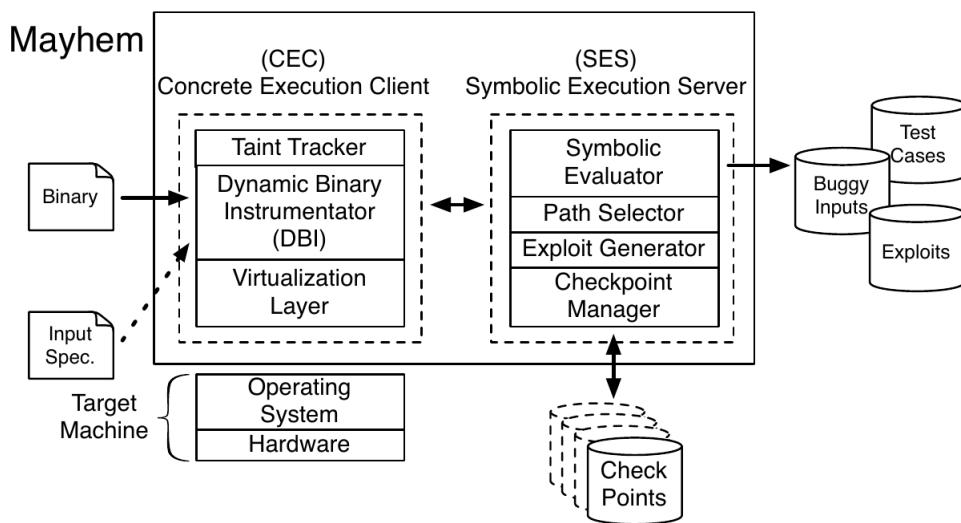


FIGURE 7.1 – Architecture des composants de Mayhem

**CEC** Le CEC gère les différentes exécutions concrètes simultanées. Chacun de ses états d'exécution concrets contient les valeurs des registres, la mémoire et l'état de l'OS (snapshot du système de fichier, du réseau et du noyau). Sous la supervision du SES et du sélecteur de chemins le CEC intervertisse la configuration concrète courante en fonction de la configuration symbolique courante. Le CEC partage au maximum les états identiques entre les exécutions concrètes lors d'un snapshot (*probablement avec du copy-on-write ou du hash-consing*).

#### Couche de virtualisation

La couche de virtualisation utilisée dans le CEC pour chaque état concret assure qu'aucun effet de bord d'une exécution ne viendra perturber une autre exécution. En effet, ceux-ci sont tous émulés par le système hôte pour conserver un état “courant” de l'OS.

**SES** Le SES contient tous les états symboliques pour chaque chemin. Un état symbolique se compose des registres symboliques et de la mémoire symbolique. À chaque *fork* d'un chemin, l'état est dupliqué via une structure immutable optimisée par *copy-on-write*. Le sélecteur de chemin (*path selector*) gère une file d'attente priorisée pour sélectionner la configuration qui va être traitée. La sélection se base sur trois heuristiques qui modulent la priorité :

1. le chemin va couvrir du code non exploré
2. le chemin a fait découvrir de nouveaux pointeurs symboliques
3. le chemin manipule un pointeur d'instruction symbolique (*très haute priorité*)

Le SES implémente le *Preconditioned Symbolic Execution* qui permet à un utilisateur de spécifier des contraintes sur les entrées comme la taille ou un préfixe fixé. Tout chemin contrevenant à ces contraintes est donc jeté.

**Optimisations de formules** `Mayhem` utilise différentes optimisations de formules dont des simplifications algébriques d'expressions, un cache de contre-exemples, du *constraint subsumption* [138] ou encore de l'analyse de teinte pour n'exécuter que les instructions teintées. Cette dernière apporte une accélération moyenne de x8. Les expérimentations ont montré que 95% des instructions d'un programme ne sont pas teintées. En sus, toutes les formules peuvent être résolues avec le mode incrémental de Z3.

### 7.2.3 Déroulement d'une analyse

Une campagne de DSE se déroule selon les étapes suivantes :

1. lancement de `Mayhem` avec comme source d'entrées symboliques un fichier, des variables d'environnement, ou une socket
2. le CEC charge le programme, se connecte au SES à partir duquel il charge les entrées symboliques (*pour la propagation de teinte*) et commence à s'exécuter
3. lorsque le CEC rencontre une condition de branchement teintée, il suspend l'exécution concrète et envoie les informations au SES
4. le SES reçoit les instructions teintées, les “lifte” en BAP (*l'IR utilisée*) puis les exécute symboliquement (*toute information concrète est demandée au CEC toujours suspendu*). Le SES maintient la formule de prédicat de chemin et la formule d'exploitabilité
5. si le ou les chemins découverts indiquent que l'exécution doit être dupliquée (*forkée*), les chemins sont envoyés au *path selector* qui va prioriser les configurations pour en choisir une, puis avertir le CEC qui va restaurer la configuration pour continuer l'exécution. Si le nombre de configurations en attente atteint le seuil, le manager de *checkpoints* génère un nouveau *checkpoint* au lieu de *forker*
6. Durant l'exécution, le SES change les contextes entre les executeurs et le CEC prend soin de sauvegarder et restaurer les états d'exécution sur la machine invitée sur laquelle tourne le programme.

7. Quand **Mayhem** détecte un saut teinté, il tente de construire une formule d’exploitabilité et vérifie auprès du solveur SMT qu’elle satisfaisable
8. Ce processus se répète jusqu’à ce qu’un bug exploitable soit trouvé, qu’un *timeout* expire, ou que tous les chemins soient couverts

Durant l’analyse, **Mayhem** cherche soit des pointeurs d’instruction teintés soit des chaînes de format teintées à partir desquelles il est en mesure de générer des exploits de type *jump-to-register* [169].

### 7.2.4 Exécution concolique “hybride”

L’exécution symbolique dynamique hybride (noté DSE-H par la suite) vise à alterner le mode *online* et le mode *offline* sur la base d’un seuil d’utilisation mémoire au-delà duquel le mode online (initialement lancé) est suspendu au profit du mode offline. Cette alternance vise à trouver le bon équilibre entre mémoire et rapidité. En effet, le mode *online* est particulièrement rapide car il permet de *forker* deux interpréteurs à un point de branchement, mais cette déduplication a forcément un coût en mémoire. Bien que plupart des exécuteurs utilisent des optimisations agressives de *copy-on-write*, les déuplications mènent inexorablement à une pénurie mémoire. L’approche de **Mayhem** est donc de démarrer dans ce mode et lorsqu’un seuil est rencontré, il passe en *offline* et au lieu de forker, crée des *checkpoints* qui seront réexécutés plus tard. Ces points de contrôle contiennent l’état symbolique (prédicat de chemin et contexte symbolique) ainsi que les informations de rejet (notamment les entrées). Lorsqu’un point est restauré, le programme est réexécuté jusqu’à ce point (sans exécution symbolique), puis l’état symbolique est restauré.

### 7.2.5 Modèle mémoire basé sur les index

Le modèle de la mémoire “*Index-based*” est représenté sous la forme d’un dictionnaire  $\mu : \mathcal{I} \rightarrow \mathcal{E}$  où les indices  $\mathcal{I}$  sont des entiers 32 bits et  $\mathcal{E}$  est une expression symbolique. **Mayhem** concrétise systématiquement les écritures mais autorise les lectures symboliques. Cette structure se heurte aux index symboliques (empiriquement rencontrés dans 40% des cas). Pour cela, **Mayhem** introduit les *memory objects* noté  $\mathcal{M}$ . Lors d’une lecture symbolique, le sous-ensemble  $\mathcal{M}$  de  $\mu$  contenant tous les index que peuvent prendre la lecture symbolique est créé. Si la lecture symbolique peut prendre toutes les valeurs ( $2^{32}$ ), cela revient à dupliquer  $\mu$  à chaque lecture symbolique. Ce fonctionnement se base sur la supposition selon laquelle un pointeur ne peut généralement prendre que quelques valeurs. **Mayhem** introduit donc des mécanismes pour s’assurer que ce soit le cas, ou bien concrétiser la lecture. Pour cela, il cherche à déterminer les bornes de valeurs du pointeur avec une recherche dichotomique via des requêtes au solveur SMT. Cela est coûteux en temps et les valeurs que peut prendre un pointeur sont rarement contiguës, ce qui casse potentiellement la correction d’une telle approche. Le papier introduit donc une solution qui utilise une analyse de **VSA** [18] par interprétation abstraite (cf. 3.1). Cette analyse ne renvoie plus un intervalle [min, max] mais un ensemble d’intervalles représentant les

valeurs pouvant être prises par le pointeur (appelé “*strided interval*”). Le résultat est toujours sur approximé mais néanmoins plus précis. Les auteurs poussent encore plus loin la technique en encodant ces intervalles sous forme d’arbres binaires facilitant la recherche, eux-mêmes “linéarisés” dans des “*bucket*” de la forme  $y = \alpha x + \beta$  (cf. article). La linéarisation se base sur le constat que les index possibles d’un pointeur s’expriment souvent sous la forme d’équations linéaires (e.g : switch, conversion unicode etc).

En définitive, **Mayhem** conserve les index de lecture symbolique si le cardinal est inférieur à 1024. Dans le cas contraire, il effectue les tests suivants :

1. teste la possibilité de rediriger le pointeur vers une zone non mappée. Si c'est le cas, il génère un cas de test de crash
2. teste la possibilité de rediriger le pointeur vers des données symboliques. Si c'est le cas, il concrétise l'index vers cette région et continue le DSE
3. sinon concrétise le pointeur

Ces trois tests permettent de gérer le cas où un premier pointeur est d’abord déréférencé, puis un second. En ayant appliqué le deuxième test, il aura été possible de rediriger le premier pointeur vers une zone contrôlée pour ensuite déclencher un bug sur le deuxième. *Cette technique est particulièrement astucieuse pour maximiser les chances de déclencher des bugs.*

#### Attention

Certains aspects de la gestion mémoire restent flous. **Mayhem** semble segmenter la mémoire sous forme de régions dans lesquelles il est en mesure d’évaluer la complexité des contraintes appliquées. Or, rien ne décrit cet aspect. Aussi, il n'est jamais explicitement fait mention de la théorie des tableaux SMT. Au vu du fonctionnement du modèle mémoire, il semblerait que **Mayhem** ne l'utilise pas car tous les calculs sont faits par le moteur de DSE directement.

### 7.2.6 Conclusion

**Limitations** Comme la quasi-totalité des moteurs de DSE, **Mayhem** ne supporte que les programmes *userland* et n'est en mesure d'analyser qu'un seul thread à la fois. Néanmoins, sa plus grosse limitation provient de la dépendance totale à l'analyse de teinte qui conditionne entièrement les instructions exécutées symboliquement. Ainsi la plateforme est dépendante des phénomènes d'*undertainting*, *overtainting* et les phénomènes de flot implicite [199].

Malgré ces quelques points négatifs, le papier est dans l’ensemble d'une limpidité exemplaire, toutes les techniques et optimisations présentées s'avèrent irréprochables aussi bien conceptuellement qu'expérimentalement.

Bien que publié 4 ans après KLEE et fortement inspiré de celui-ci, **Mayhem** reste pionnier dans le domaine de l'exécution symbolique dynamique binaire appliquée pour de la recherche de vulnérabilités. C'est aussi l'une des premières publications à proposer diverses

optimisations apportant un vrai passage à l'échelle de la technique. Cette analyse s'est évidemment focalisée sur l'état de développement de **Mayhem** tel qu'il était en 2012 et il ne fait aucun doute que celui-ci a depuis considérablement évolué sous l'impulsion du **CGC** grâce auquel il est devenu le meilleur **CRS** existant en gagnant le concours (*les CRS sont décrits dans la section 12.4*). L'avenir de l'outil est très prometteur car David Brumley, le fondateur, a annoncé<sup>65</sup> le 3 avril avoir effectué une levée de fond de 15M\$ pour développer **Mayhem** comme une plateforme SAS, ou intégrée dans le cycle de développement de test et d'intégration de logiciel tiers.

Concernant PASTIS, cet outil est néanmoins *de facto* écarté, car il n'est pas disponible publiquement. Néanmoins, l'architecture de fonctionnement et les optimisations employées ont clairement influencé les propositions effectuées au Chapitre 17.

---

65. <https://forallsecure.com/blog/2019/04/09/onward-to-the-next-chapter-in-a-forallsecure-journey/>

## 7.3 DSE #2 : Manticore

Le code de l'outil **Manticore** a été publié en avril 2017 par [Trail of Bits](#) (ToB) qui le développe. Il a aussi été développé dans le contexte du challenge [CGC](#). Depuis, il est très activement développé et comptabilise plus de 729 commits et 68 contributeurs, dont au moins 5 développeurs actifs.

Il a été choisi, car il est activement supporté, car il supporte les architectures ARM, mais aussi, car son design et son implémentation semblent être faciles et rapides à prendre en main.

Aucune publication académique ou industrielle ne détaille le fonctionnement de l'outil. Seuls plusieurs posts de blog discutent de certaines fonctionnalités techniques<sup>66 67</sup>. Ainsi, la plupart des informations présentées ci-dessous ont été glanées en examinant directement le code source de l'application.

### 7.3.1 Caractéristiques fonctionnelles

**Manticore** est entièrement développé en Python et requiert la version 3.6. La version utilisée dans cet état de l'art est la 0.2.5. L'outil se présente comme un outil en ligne de commande pour des cas d'utilisation prédefinis, mais la plus-value de **Manticore** se situe dans l'API Python exposée permettant de *hooker* et moduler programmatiquement l'exécution symbolique. Le moteur permet d'utiliser nativement plusieurs coeurs pour effectuer la couverture en parallèle (*grâce au module multiprocessing*). Les trois principales dépendances sont [capstone](#)<sup>68</sup> pour le désassemblage des instructions, [unicorn](#)<sup>69</sup> pour l'émulation et [pyelftools](#)<sup>70</sup> pour le chargement des fichiers ELF.

En ce qui concerne les plateformes et architectures, **Manticore** supporte les configurations suivantes :

- Bytecode Ethereum Virtual Machine (EVM) (smart-contract de la blockchain Ethereum) ;
- Architectures x86, x86\_64 et ARMv7 pour Linux ELF.

Il ne supporte donc pas ARMv8, ce qui est fortement limitant si la carte de test choisie utilise cette architecture. En outre, voici quelques caractéristiques de l'outil :

- possède un cache d'instructions décodées (*ne re désassemble pas deux fois la même adresse*). L'inconvénient est que l'outil ne supporte par conséquent pas le code automodifiant ;

---

66. <https://blog.trailofbits.com/category/manticore/>

67. <https://blog.trailofbits.com/2019/04/01/performing-concolic-execution-on-cryptographic-primitives>

68. <http://www.capstone-engine.org/>

69. <http://www.unicorn-engine.org/>

70. <https://github.com/eliben/pyelftools>



FIGURE 7.2 – Logo Manticore

- transfert directement la sémantique des instructions dans leur représentation symbolique (*aucune IR n'est utilisée*) ;
- la plupart des instructions flottantes ne sont pas modélisées (en particulier `fxrstor`, `fxtsave`) ;
- a introduit récemment la fonctionnalité de *state-merging*<sup>71</sup> via un *plugin*<sup>72</sup>.

#### Attention

Manticore est un DSE où tout est émulé, les syscalls, le système de fichier etc. Dans la taxonomie des exécuteurs symboliques, il est donc plutôt statique.

### 7.3.2 Stratégies de couverture

La stratégie de couverture est l'un des seuls paramètres bas-niveau qui peut directement être configuré via la ligne de commande avec `-policy` (*à ne pas confondre avec les politiques de concrétisation ci-dessous*). La ligne de commande propose les stratégies `random`, `adhoc`, `uncovered`, `dicount`, `icount`, `syscount` et `depth`. La plupart ne sont cependant pas implémentées. En pratique, les seules existantes sont : `random`, `uncovered` et `branchlimited` dans `manticore/core/executor.py`.

### 7.3.3 Gestion de la mémoire

#### Modèle mémoire

Toute la mémoire est gérée par une classe `Memory`, qui effectue les différentes allocations et désallocations qui sont faites durant l'exécution. Dérivée de cette classe, `SMemory` gère une représentation symbolique de la mémoire, et en particulier un mapping des pages à des offsets symboliques. Enfin, un modèle mémoire expérimental, `LazySMemory`, dérivé de la précédente classe, permet une gestion complètement symbolique de la mémoire avec des index symboliques. Dans ces trois modèles, chaque fichier, chaque segment, la pile et toutes les pages mémoires alloués avec `mmap` se voient attribués une mémoire virtuelle gérée par la classe. Chacune de ces mémoires possède un tableau dans la théorie des tableaux SMT. Ce modèle est très intéressant, car il permet un bien meilleur passage à l'échelle des formules sur les tableaux (*ceux-ci étant individuellement bien moins contraints*).

#### Concrétisations

Les concrétisations des adresses mémoires, et plus généralement des expressions, se fait à partir de 4 politiques de concrétisation différentes qui sont :

71. <https://blog.trailofbits.com/2019/01/25/symbolic-path-merging-in-manticore/>

72. <https://github.com/trailofbits/manticore/blob/13e3ab61acb2bf184ac853cfb63fb54\655d13b00/manticore/core/plugin.py#L314>

- **MINMAX** : Concrétise en cherchant les bornes min et max de la valeur ;
- **MAX** : Concrétise et renvoie la valeur maximum ;
- **MIN** : Concrétise et renvoie la valeur minimum ;
- **SAMPLED** : Renvoie un échantillon de valeurs possibles ;
- **ONE** : Renvoie une valeur possible ;
- **ALL** : Renvoie toutes les valeurs possibles.

La minimisation et maximisation de valeur ne sont pas implémentées manuellement, mais tirent parti des fonctions d'objectifs de Z3 qui optimise la résolution de la formule dans ce but<sup>73</sup>. Ceci rend le mécanisme dépendant de Z3, mais reste certainement plus efficace que s'il avait été fait à la main avec une recherche dichotomique en  $\log(n)$ .

#### Attention

Les concrétisations dans le contexte de **Manticore**, qui effectue de l'exécution purement symboliquement, reviennent à faire des appels au solveur pour fixer des valeurs symboliques à des valeurs constantes. Ces valeurs ne sont en rien représentatives d'une exécution concrète (*comme c'est le cas pour du vrai DSE*).

### 7.3.4 Modélisation du système et des entrées

#### Gestion des entrées symboliques

Les sources d'entrées symboliques supportées sont les suivantes :

- ligne de commande (**argv**) ;
- variable d'environnement (**envp**) ;
- l'entrée standard (**stdin**) ;
- fichier symbolique (dont on fournit le nom et éventuellement le contenu avec des caractères jokers) ;
- sockets (*dont le support est expérimental et limité à 64 octets*).

Un atout de la gestion des entrées est la possibilité de fixer certains caractères, tout en spécifiant d'autres comme étant des jokers (*wildcards*). Cela permet d'éviter au solveur de perdre du temps à trouver des valuations pour des octets superfétatoires à l'entrée.

#### Gestion des appels de bibliothèques

Une API complète est fournie permettant de redéfinir le comportement d'une fonction par une fonction Python. Pour cela, il suffit d'implémenter le comportement symbolique de la fonction en Python, puis de hooker l'appel à la fonction originale pour le rediriger vers l'implémentation en Python. Le fichier *manticore/native/models.py* fournit deux exemples d'implémentation symbolique pour les fonctions **strcmp** et **strlen**.

---

73. <https://rise4fun.com/Z3/tutorialcontent/optimization>

## Gestion des appels système

**Manticore** simule entièrement le Linux sur lequel est exécuté symboliquement le programme. Un travail titanique a donc été effectué pour simuler le chargement d'un programme et un grand nombre d'appels système. Tout cela est géré par la classe `Linux` dans `manticore/platforms/linux.py`. Les fichiers, les dossiers, les sockets tout est simulé via des classes. Par-dessus cela, la classe `SLinux` hérite de `Linux` pour fournir plusieurs fonctionnalités comme la création de fichiers/dossiers symboliques, etc. Cela permet théoriquement d'ouvrir, lire et écrire des fichiers ou sockets dont on ne connaît même pas le nom, voire même de résoudre des contraintes dessus.

### 7.3.5 Résolution des formules

#### Logiques et Théories

La résolution de formules se base sur Z3 [257], interfacé directement sur le binaire via le module `subprocess` de Python. Aucun usage de l'API Python n'est donc fait, et Z3 étant lancé en mode interactif, la communication s'effectue donc via `stdin`. Ce design permet théoriquement de s'interfacer facilement avec d'autres solveurs, cependant seul Z3 est supporté ce qui rend ce choix injustifié. Cette interface se trouve dans `manticore/core/smtlib/solver.py`.

La logique utilisée est QF\_AUFBV (cf. 6.8.3), elle utilise des formules de bit-vecteurs et tableaux avec fonctions non interprétées (UF). N'utilisant aucune API, toutes les opérations, expressions et types SMTLIB2 sont définis dans des classes *ad-hoc*. L'avantage de tout redéfinir est de pouvoir ajouter des attributs aux expressions, et notamment un booléen indiquant s'il est teinté ou non.

#### Note

Manticore utilise la logique QF\_AUFBV, mais rien ne semble utiliser des fonctions non interprétées. L'usage en est d'ailleurs rarement justifié pour de l'analyse de code binaire. Il serait plus juste pour Manticore d'utiliser QF\_ABV.

## Optimisation de formules

Un visiteur de formules est implémenté permettant de faire des simplifications de constantes et d'opérations arithmétiques. La simplification de constantes est discutable, car elle est implémentée entièrement en Python à base de `isinstance` alors qu'un solveur effectue aussi ce genre d'opérations certainement plus efficacement. La classe `ArithmeticSimplifier` effectue plusieurs simplifications sur les extractions, concaténations, mais aussi sur les `select` dans les tableaux. Elle vise à simplifier un index de lecture en itérant la chaîne de `store` jusqu'à trouver le même index (voir Optimisation des formules

6.8.5). Le Listing 15 montre l’implémentation de cette simplification qui, à l’inverse de l’implémentation de Binsec [119], est malheureusement quadratique en complexité. Cet algorithme est potentiellement quadratique en le nombre de `select` de l’expression et de `store` de la formule. Fortuitement, le modèle mémoire de Manticore atténue la taille des tableaux mémoires (*qui ne représentent qu’un fragment mémoire*).

---

#### Listing 15 Optimisation de `select` dans Manticore

---

```

1 def visit_ArraySelect(self, expression, *operands):
2     ''' ArraySelect (ArrayStore((ArrayStore(x0,v0) ...),xn, vn), x0) -> v0 '''
3     arr, index = operands
4     if isinstance(arr, ArrayVariable):
5         return
6
7     if isinstance(index, BitVecConstant):
8         ival = index.value
9
10    while isinstance(arr, ArrayStore) and isinstance(arr._operands[1], \
11                  BitVecConstant) and arr._operands[1]._value != ival:
12        arr = arr._operands[0] # arr.array
13
14    if isinstance(index, BitVecConstant) and isinstance(arr, ArrayStore) and \
15        isinstance(arr.index, BitVecConstant) and arr.index.value == index.value:
16        return arr.value
17    else:
18        if arr is not expression.array:
19            return arr.select(index)

```

---

### 7.3.6 Exécution des tests

#### Compilation des suites de test

Cet outil souffre de deux inconvénients. D’une part, il supporte uniquement ARMv7, et d’autre part, l’exécution symbolique de programmes chargés dynamiquement est extraordinairement lente. Ceci s’explique par le fait que le chargement et l’exécution de `ld-linux-x86-64.so.2` sont entièrement émulés (et tout le mapping en mémoire est répercuté dans les classes internes. Pour l’exécution des benchmarks de tests atomiques il a donc fallu compiler la suite de test avec les commandes du listing 16.

#### Exécution d’une analyse

Manticore fournit un programme du même nom permettant de lancer l’exécution symbolique d’un programme avec une accommmandante facilité. Les fichiers de test générés

**Listing 16** Compilation de la suite de tests pour Manticore

```

1 # For the ARMv7 build
2 CC=arm-linux-gnueabi-gcc cmake -DTARGET_ARCH=ARM -DSTATIC=1 ..
3 make
4
5 # For the LAVA ARMv7 compilation
6 CC=arm-linux-gnueabi-gcc LDFLAGS="-static" ./configure --host=arm-linux-gnueabi \
7                               --build=arm-linux --enable-install-program=base64
8 make

```

**Adaptation des tests atomiques**

Tous les tests atomiques utilisent la fonction `fgets` pour transmettre la chaîne d'entrée à la fonction de test. Or, cette fonction fait intervenir l'appel système `sys_fseek` qui semblait bugué lors des tests. Pour **Manticore**, il a donc fallu adapter tous les tests pour utiliser la fonction `read`.

sont créés dans un dossier dût de *workspace*, au nom aléatoire préfixé par `mcore_`, contenant les fichiers suivants pour chaque cas de test :

- `test_00000XX.argv` : ligne de commande du programme ;
- `test_00000XX.env` : variables d'environnement du programme ;
- `test_00000XX.input` : ensemble des entrées du programme ;
- `test_00000XX.messages` : état du programme (registres, mémoire) à la génération du test ;
- `test_00000XX.net` : variables d'entrées d'une socket ;
- `test_00000XX.pkl` : état mémoire du chemin (*permet d'interrompre et de reprendre le SE ultérieurement*) ;
- `test_00000XX.smt` : formule SMT utilisée pour générer les entrées ;
- `test_00000XX.stout` : sortie standard ;
- `test_00000XX.stderr` : sortie d'erreur ;
- `test_00000XX.stdin` : entrées du programme générées ;
- `test_00000XX.syscalls` : appels système ;
- `test_00000XX.trace` : trace d'exécution des adresses.

Pour chaque test généré il donc possible de réexécuter le programme avec le `argv`, `env`, `net` et `stdin` générés. Le dossier contient aussi le fichier `command.sh` contenant la commande utilisée pour lancer les tests.

**Mécanisme de plugin**

L'architecture de **Manticore** permet la création de plugins qui viennent s'intégrer dans l'exécution via des *hooks* et callbacks activés lors de divers événements comme la lecture, l'écriture d'un registre ou d'une adresse mémoire. D'autres callbacks permettent de gérer

les chargements des états ou la création des cas de tests. À titre d'exemple, voici le script (*simplifié*) utilisé pour exécuter symboliquement chacun des tests atomiques de la suite de test, sachant que TRIGGERED renvoie 1 comme code de retour du programme.

---

**Listing 17** Script d'exécution des tests atomiques pour Manticore

---

```
1 from manticore.native import Manticore
2 from manticore.core.plugin import Plugin
3
4 TRIGGER = 1
5 NOT_TRIGGERED = 0
6
7 class BenchmarkPlugin(Plugin):
8     def will_generate testcase_callback(self, state, testcase, msg):
9         exitcode = int(msg.split(" ")[-1])
10        if exitcode == TRIGGER:
11            print("Bomb triggered !")
12            self.manticore.shutdown()
13            sys.exit(TRIGGER)
14
15 if __name__ == '__main__':
16     binary_name = sys.argv[1]
17     workspace = "mcore_" + binary_name
18     m = Manticore(binary_name, workspace_url=workspace)
19     m.register_plugin(BenchmarkPlugin(workspace))
20     m.run()
21     sys.exit(NOT_TRIGGERED)
```

---

### 7.3.7 Conclusion

La logique derrière l'implémentation de **Manticore** était d'avoir un outil de DSE sans trop de fonctionnalités, mais qui soit très facile à prendre en main, très fonctionnel et utilisable dans toutes les situations. Force est de constater que cela est réussi. L'utilisation comme la compréhension et l'appropriation du code semble simple et rapide. Il a très rapidement fourni des résultats pour les tests atomiques et peut être utilisé sans connaissance de l'exécution symbolique.

Le revers de la médaille est un code écrit avec négligence, contenant pas moins de 72 “TODO” et 45 “FIXME”. Le code semble grandement en chantier. Les benchmarks ont dû être adaptés pour utiliser `read`, et de nombreux autres sont venus ternir les résultats des benchmarks. **Manticore** ne jouit donc pas d'une très bonne stabilité. Un dernier point limitant pour PASTIS est la lenteur de l'exécution symbolique. Comme absolument tout est simulé, charger un programme dynamiquement prend déjà un temps considérable.

Le support des sockets, bien qu'expérimental, est néanmoins fonctionnel, ce qui serait profitable pour le démonstrateur. De plus, la prise en main et l'appropriation rapide de l'API rendraient cet outil tout à fait adaptable aux besoins du projet, bien qu'aucune connexion naturelle n'ait été pensée pour du fuzzing.

## 7.4 DSE #3 : KLEE

Cet outil a été choisi, car c'est indéniablement le plus ancien encore maintenu et activement développé. De la même manière que AFL, il a profité de nombreuses contributions de la communauté au cours du temps et fourni un large panel de fonctionnalités. Fonctionnant à partir des sources sur de l'IR LLVM, cet outil est naturellement un très bon candidat pour le projet PASTIS.

KLEE a été initié par Cristian Cadar en 2008 [56] et il est toujours activement maintenu par celui-ci depuis plus de 11 ans. Sur cette période, les développements ont été continus. Le projet jouit d'une vivacité exemplaire (cf. graphe d'activité 16.3) garante du support et de la pérennité du projet maintenant supporté par le *Imperial College of London*.

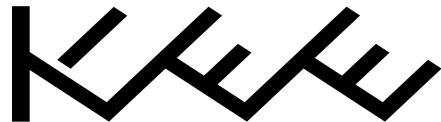


FIGURE 7.3 – Logo KLEE

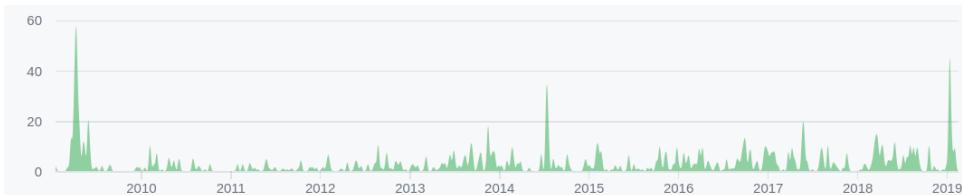


FIGURE 7.4 – Graphe d'activité des commits de KLEE

KLEE étant l'exécuteur symbolique dynamique le plus ancien et le plus développé, il a suscité un grand nombre de publications. L'étude ci-dessous se base sur la publication fondatrice [56], le code source disponible sur Github<sup>74</sup> et la documentation assez didactique du site<sup>75</sup>.

La version de KLEE analysée est la version 2.1-pre basée sur LLVM 6.0.1. Le code, développé majoritairement en C++, est relativement compact comparé au large panel de fonctionnalités de l'outil.

### 7.4.1 Fonctions intrinsèques

L'API fournit un certain nombre de fonctions appelées *intrinsic* qui peuvent être ajoutées manuellement dans le code pour configurer et guider l'exécution. En voici quelques-unes :

- `klee_make_symbolic` : transforme la variable donnée en paramètre symbolique (*même si elle n'est pas directement dérivée d'entrées*)
- `klee_assume` : équivalent d'un `assume`, mais dans le prédicat logique

74. <https://github.com/klee/klee>

75. <https://klee.github.io>

- `klee_prefer_cex` : indique à KLEE de préférer un certain range de valeurs lors de la génération de cas de tests (*e.g. ascii*).
- `klee_range` : créer une valeur symbolique avec un range de valeurs données (*donc valeur symbolique contrainte*).

C'est notamment grâce à ces fonctions qu'il est possible d'aider l'exécution à passer à l'échelle [127] (voir. Section Analyse semi-automatisée 6.9). Toutes les fonctions sont définies dans `klee/include/klee/klee.h`.

### 7.4.2 Gestion des entrées symboliques

Conceptuellement, pour toutes les entrées du programme comme la ligne de commande, les variables d'environnement et les appels système (accès fichiers, réseau etc). KLEE essaye de modéliser symboliquement toutes les valeurs qui peuvent être fournies par ces entrées. KLEE modélise certains appels système via des modèles écrits en C, au nombre de 40 sur un total de 2500 lignes de code. Lorsqu'intervient un appel système (en particulier les accès fichiers), si les paramètres sont entièrement concrets, l'appel est transmis à l'OS, et si ceux-ci sont en partie symboliques le modèle associé est appelé pour modéliser symboliquement les effets de bord. Pour les appels de bibliothèques comme la `libc`, KLEE utilise généralement `uClibc`, une implémentation minimale beaucoup plus légère que celle de GNU.

### 7.4.3 Gestion de la mémoire

Le modèle mémoire utilisé est basé sur des "objets mémoire", qui représentent des fragments indépendants de mémoire, auxquels sont associés des tableaux dans la théorie des tableaux. Ainsi, chaque allocation, segment, etc possède un objet mémoire associé. Ce modèle permet un passage à l'échelle par rapport à un modèle d'une mémoire à plat (*flat*).

Lorsqu'un pointeur référence plusieurs objets, KLEE *fork* le chemin en plusieurs états correspondants au pointeur référençant chacun des objets mémoire. Cette méthode est potentiellement très coûteuse en nombre de forks si le pointeur possède un large ensemble de cibles "*points-to set*" mais en pratique, les auteurs indiquent que c'est rarement le cas.

Par ailleurs, KLEE gère les objets mémoire avec une structure de Copy-on-Write (CoW) permettant de fork sans coût mémoire immédiat.

### 7.4.4 Gestion des stratégies de recherche

Par contributions successives au projet, plusieurs stratégies de recherche ont été introduites. Les deux stratégies documentées dans le papier de recherche sont :

- **Sélection aléatoire**, (*random-search*) : choisit au hasard l'état associé à un chemin dans l'arbre de tous les chemins découverts jusqu'alors. Cette stratégie garantit une probabilité de sélection équilibrée entre les états (éitant de rester bloqué dans des boucles ou autre)
- **Optimisé pour la couverture**, *Coverage-Optimized Search* : Sélectionne les états les plus disposés à découvrir du nouveau code dans le futur grâce à une métrique de poids.

Chacune de ces deux stratégies sont appliquées l'une et l'autre successivement (*avec un round-robin*).

#### 7.4.5 Résolution des formules & Optimisations

Les formules générées sont dans la logique QF\_AUFBV. Historiquement, KLEE se base sur STP [129] mais il supporte aussi Z3 [257], et maintenant metaSMT [296] qui ajoute tous les solveurs sous-jacents dont Boolector [50]. Différentes optimisations sont effectuées sur les formules et une structure de hash-consig permet de jouer le rôle de cache de formules.

Le coût de résolution des formules domine tous les autres coûts lors du DSE, KLEE implémente donc les optimisations ci-dessous :

**Réécriture d'expression** : *Expression Rewriting* pour les opérations arithmétiques sur des constantes,  $x \times 2^n \rightarrow x \ll n$ , etc.

**Simplification des contraintes** : (de  $\varphi$ ) *Constraint Set Simplification*. Le calcul d'un prédicat de chemin ne fait que rajouter des contraintes de plus en plus précises. Les précédentes sont alors simplifiées. Par exemple,  $x < 10$ , puis plus tard la contrainte  $x = 5$  mène à la simplification de la première contrainte,

**Concrétisation déduite** : *Implied Value Concretization* Remplace des expressions par leurs valeurs concrètes lors d'écriture en mémoire pour simplifier une lecture ultérieure par la valeur constante. Par exemple,  $x + 1 = 10$  écrit à une adresse mémoire peut être simplifié par y écrire 9.

**Contraintes indépendantes** : *Constraint Independence* Sur un prédicat de chemin, conserve les différents sous-ensembles indépendants les uns des autres et n'utilise que ceux impliqués dans la résolution de la formule finale. Deux formules sont disjointes si leurs symboles utilisés (*variables libres*) sont disjoints.

**Cache de contre-exemples** : *Counter-example Cache* conserve un cache de formules redondantes en gardant pour chacune une valuation des symboles formant un contre-exemple. Ce cache se présente sous la forme d'un ensemble dont chaque élément est un ensemble de formules, encodées dans une structure de données dérivée de UBTree[175] permettant une recherche rapide aussi bien d'un sous-ensemble que d'un sur-ensemble d'une formule (*constraint set*).

### 7.4.6 Compilation et utilisation

En suivant la documentation concernant la compilation du projet<sup>76</sup>, compiler KLEE est surprenamment facile vu la taille et les dépendances du projet.

#### Exécution d'un programme

Le test d'un programme s'effectue de deux manières (pouvant s'entremêler). La première est de compiler et escamoter (*linker*) le programme avec le *runtime* de KLEE (e.g commande ci-dessous). Le second est de compiler le programme de manière indépendante.

Le premier mode de KLEE (standard) est d'ajouter les directives `klee_make_symbolic` sur toutes les variables symboliques du programme puis de le compiler en bytecode LLVM avec la commande :

```
clang -I klee/include -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone file.c
```

---

Pour les programmes composés de plusieurs fichiers, `wl1vm`<sup>77</sup> (Whole-program LLVM) permet en le spécifiant comme compilateur à la chaîne de compilation (\$CC) de compiler tout un programme dans un fichier .bc.

Dans le premier mode avec les annotations, il suffit ensuite d'exécuter KLEE sur le bytecode LLVM généré. Dans le second mode, l'argument `-posix-runtime` est indispensable, sans quoi KLEE sera incapable d'injecter les entrées symboliques lors de leur récupération dans le programme. La commande ci-dessous donne un exemple de l'exécution de KLEE dans le second mode.

```
./klee/build/bin/klee --libc=uclibc --posix-runtime file.bc --sym-stdin 1024
```

---

À noter qu'en plus de `-posix-runtime`, il est nécessaire d'indiquer quelles sont les sources d'entrées symboliques et leur taille – ici `stdin` avec une taille de 1024. D'autres options permettent de spécifier d'autres sources d'entrées symboliques<sup>78</sup>.

---

76. <http://klee.github.io/build-llvm60/>

77. <https://github.com/travitch/whole-program-llvm>

78. <https://klee.github.io/docs/options/>

## Dossier de workspace

Une fois le bytecode obtenu il suffit d'exécuter `klee file.bc` pour lancer l'exécution symbolique du programme. Avec les paramètres par défaut ceci est très simple à effectuer. L'exécution génère un dossier contenant les fichiers suivants<sup>79</sup> :

- assembly.ll : bytecode LLVM au format texte ;
- info : informations concernant l'exécution (temps d'exécution, nombres de chemins couverts etc) ;
- messages.txt : messages de sortie de KLEE ;
- warnings.txt : messages d'alerte émis par KLEE ;
- run.stats : fichier de statistiques utilisé par `klee-stats` pour afficher les informations de couverture, le temps passé par le solveur et d'autres statistiques avancées. `klee-stats` permet notamment de servir ces informations via une API REST compatible avec grafana<sup>80</sup>, etc ;
- run.istats : fichier de statistiques sur la couverture de chaque ligne du programme. Le format est celui utilisé par Kcachegrind<sup>81</sup> ;
- test0000XX.ktest : fichier de test pour chaque chemin découvert. Ces fichiers sont lisibles par `ktest-tool` permettant d'afficher le contenu.

D'autres fichiers peuvent être générés en fonction des options fournies à KLEE.

## Rejou des tests

Le choix du mode d'utilisation de KLEE a surtout un impact sur la manière de rejouer les tests générés. Dans le premier, l'exécution des tests générés sur la cible passe par la compilation du programme avec la bibliothèque `libkleeRunttest` qui va remplacer les `klee_make_symbolic` par les entrées du ktest envoyé en variable d'environnement (cf. 22).

---

### Listing 18 Commandes de compilation et d'exécution d'un cas de test

---

```
1 export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:klee/build/lib/
2 gcc -I klee/include -L klee/build/lib/ -lkleeRunttest -o test test.c
3 KTEST_FILE=klee-last/test0000XX.ktest ./test
```

---

### 7.4.7 Exécution des benchmarks LAVA-M

La compilation de la suite LAVA-M pour KLEE s'est très fortement inspiré d'un tutoriel expliquant comment compiler les coreutils pour les tester avec KLEE<sup>82</sup>. Cela passe notamment par l'utilisation de `wllvm` comme compilateur, qui s'occupe de compiler chaque

79. <http://klee.github.io/docs/files/>

80. <https://grafana.com/>

81. <https://kcachegrind.github.io/html/Home.html>

82. <https://klee.github.io/tutorials/testing-coreutils/>

utilitaire en bytecode LLVM, puis ensuite regrouper les différentes dépendances en un seul et même fichier. Il ne reste plus alors qu'à utiliser la bibliothèque uClib et le mode POSIX pour effectuer le DSE sans avoir eu à annoter manuellement chacun des utilitaires (*donc le second mode*).

#### 7.4.8 Suite de tests

Dans la publication originale, KLEE a été testé sur les **coreutils** pour lesquels il a trouvé plusieurs bugs et atteint une meilleure couverture globale que la suite de test officielle (*notoirement exhaustive*). D'autres expérimentations ont aussi été effectuées sur **Busybox**<sup>83</sup>, sur lequel il a aussi été possible de faire de l'analyse différentielle avec les **coreutils**. Une dernière expérimentation a été faite sur le noyau HiStar développé à l'époque à Standford. Ces trois benchmarks effectués dans la publication montrent la capacité de l'outil à passer à l'échelle sur des programmes relativement conséquents, bien qu'en pratique il s'avère relativement difficile d'atteindre les mêmes résultats (cf. Benchmarks LAVA 8.2).

#### 7.4.9 Conclusion

La longévité et l'assise de KLEE dans l'écosystème des DSE en fait un candidat de premier choix dans le contexte du projet PASTIS. Ce projet a su dès 2008 faire les bons choix techniques, à savoir utiliser l'IR LLVM (alors beaucoup moins utilisée qu'aujourd'hui), et cette décision a eu un impact énorme sur son adoption dans divers contextes.

Son utilisation dans un contexte d'utilisation “standalone” avec couverture du code serait vraisemblablement un très bon choix d'autant qu'il possède aussi un certain support des connexions réseau. Le principal frein à son utilisation serait certainement la difficulté de s'interfacer avec, celui-ci étant conçu comme un programme et non pas comme une bibliothèque dont il serait possible de réutiliser des composants.

---

83. <https://busybox.net/about.html>

## 7.5 DSE #4 : Angr

Cet outil est un framework complet d'analyse binaire écrit en Python, employant à la fois des méthodes d'analyse statique et dynamique, et notamment de l'exécution symbolique. L'outil supporte plusieurs architectures et formats d'exécutables. Il est open source avec la licence BSD-2. Il est activement maintenu par les chercheurs du *Security Lab*<sup>84</sup> (seclab) à l'université de Santa Barbara. La version étudiée est la 8.18.10.25 fournie sur pip. Ce projet a donné lieu à de nombreux posts de blogs et plusieurs articles scientifiques, mais le principal, celui étudié ici, est le suivant :

*“(State of) The Art of War : Offensive Techniques in Binary Analysis” par Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna publié au IEEE Symposium on Security and Privacy, S&P 2016*

Tous les auteurs sont(étaient) à l'université de Santa Barbara et cet article a l'ISBN : 978-1-5090-0824-7 et le DOI 10.1109/SP.2016.17.

Cet outil est la pierre angulaire du CRS appelé *Mechanical Phish*<sup>85</sup> [321] utilisé par l'équipe *Shellphish*<sup>86</sup> lors du challenge CGC, pour lequel ils ont gagné la 3ème place (voir Chapitre 12, Section 12.4). C'est à l'issue de la finale de ce challenge que le code de *angr* a été publié en sources libres. Depuis, il est toujours aussi activement maintenu avec pas moins de 9682 commits et 107 contributeurs dont 4 principaux.

### 7.5.1 Fonctionnalités & Écosystème

Ce framework fournit un éventail très large de fonctionnalités, allant du désassemblage à l'analyse data-flow. L'architecture et l'API facilitent grandement l'implémentation d'analyses.

- Désassembleur ;
- Reconstruction du graphe flot de contrôle ;
- Comparaisons de binaires (diff) ;
- *Backward-slicing* ;

---

84. <https://seclab.cs.ucsb.edu>

85. <https://github.com/mechaphish>

86. <https://github.com/shellphish>



FIGURE 7.5 – Logo angr

- Analyse *data-flow*.

Une interface graphique (*Graphical User Interface (GUI)*) expérimentale appelée `angr-management` est aussi en phase de développement<sup>87</sup>.

**Modules** Le développement d'`angr` s'articule autour de différents projets auxiliaires rendant le code plus modulaire. Ces dépendances internes sont multiples, les principales sont les suivantes :

- `claripy`<sup>88</sup>, wrapper Python pour la résolution de contraintes sur différents solveurs (cf. Section 7.5.8).
- `cle`<sup>89</sup>, outil de chargement de différents formats d'exécutables (*avec les bibliothèques dynamiques associées*). Charge le programme comme le ferait `ld.so` avec résolution des imports, abstraction de la mémoire du process etc.
- `PyVEX`<sup>90</sup> : bibliothèque d'interface pour VEX en Python (cf. 7.5.2).
- `archinfo`<sup>91</sup> : bibliothèque d'abstraction d'architectures (ARM, x86), permettant d'itérer les instructions, registres etc.

**Outils basés sur angr.** Cet ensemble de projets et toutes les fonctionnalités fournies, ont notamment permis de créer des outils basés sur `angr`. Le plus notable est `Driller` [333], développé par les mêmes personnes et apportant le support du fuzzing hybride. Parmi les autres projets, il y a :

- `angrop`<sup>92</sup> : Générateur de ROP chain automatique
- `patcherex`<sup>93</sup> : Durcissement automatique de binaires (*pour DECREE*)
- `rex`<sup>94</sup> : Génération automatique d'exploits `AEG`, (*principalement pour DECREE*)

## 7.5.2 Représentation intermédiaire

Cet outil utilise la représentation intermédiaire de `Valgrind` [264], VEX. Ce choix s'est fait pour plusieurs raisons : le langage VEX supporte un grand nombre d'architectures (x86, AMD64, ARM, ARM64, MIPS). En plus de son large support d'architectures, cette représentation est particulièrement bien documentée<sup>95</sup> et est activement développée. De plus, elle a été créée pour l'analyse de programmes et l'instrumentation, rendant son utilisation aisée pour `angr`. (Cela nécessite quelques ajustements minimes, `angr` maintient un fork de la bibliothèque originale)

Le langage VEX est agnostique de l'architecture et sans effets de bords. Le langage dispose de 5 classes d'objets principaux :

---

87. <https://github.com/angr/angr-management>  
88. <https://github.com/angr/claripy>  
89. <https://github.com/angr/cle>  
90. <https://github.com/angr/pyvex>  
91. <https://github.com/angr/archinfo>  
92. <https://github.com/salls/angrop>  
93. <https://github.com/angr/patcherex>  
94. <https://github.com/shellphish/rex>  
95. [https://github.com/angr/vex/blob/dev/pub/libvex\\_ir.h](https://github.com/angr/vex/blob/dev/pub/libvex_ir.h)

- **Expressions**, représentent une valeur calculée ou une constante (lecture mémoire, lecture de registre, résultats d'opérations arithmétiques).
- **Operations**, représentent une modification d'une **Expression** (opération arithmétique entière ou flottante, opérations bit à bit).
- **Temporary variables**, représentent les registres internes du langage intermédiaire.
- **Statements**, représentent les changements d'états de la machine cible (les effets d'une écriture mémoire, ou d'un registre).
- **Blocks**, sont un groupe de **Statements**, représentant un basic-block dans l'architecture cible.

#### Support expérimental du Java

Très récemment <sup>a</sup>, le support de l'exécution symbolique du Java a été introduit dans **angr**. Celle-ci s'appuie sur le langage intermédiaire *Soot IR* <sup>b</sup>, pour lequel a été développé un binding Python dédié, **PySoot** <sup>c</sup>, permettant de *lift*er aussi bien du Java que du DEX Android. Dans le contexte d'Android, cela permet donc de faire de l'exécution symbolique “cross-langage” avec le DEX pour Java et le binaire pour le JNI. Cette interchangeabilité de langage est implémentée en interne via un *engine* capable de passer du Java au JNI et vice-versa. Quelques expérimentations effectuées à Quarkslab ont montré qu'au-delà de l'exemple montré, le *lifting* PySoot d'une application plus conséquente et son exécution ne passe définitivement pas à l'échelle. Bien que les performances ne soient pas au rendez-vous, cette initiative montre la modularité et la flexibilité de l'outil pour lequel il est possible d'ajouter un *engine* adapté à l'IR à exécuter.

- a. [https://angr.io/blog/java\\_angr](https://angr.io/blog/java_angr)
- b. <https://github.com/Sable/soot>
- c. <https://github.com/angr/pysoot>

### 7.5.3 Gestion des états

**angr** propose une interface simpliste et personnalisable pour son moteur d'exécution symbolique nommée **SimulationManager**. Cette interface permet de contrôler finement l'exécution d'un programme.

Les états de l'interface de simulation, **SimState**, sont une brique essentielle au bon fonctionnement du **SimulationManager**. Ils contiennent l'état de la mémoire du programme, les registres, les données du système de fichiers, etc. Ces états sont organisés en *stash* (groupe d'états). Ces *stashes* peuvent être filtrés, modifiés, et permettent par exemple d'explorer plusieurs états du programme en parallèle.

Ces “*stashes*” peuvent avoir les états suivants <sup>96</sup> :

- active ;

96. <https://docs.angr.io/core-concepts/pathgroups#stash-types>

- deadended ;
- pruned ;
- unconstrained ;
- unsat.

L'état par défaut est *active*.

#### 7.5.4 Exécution & Stratégies d'exploration

L'exécution d'un programme – avancer à travers les états – peut être faite selon trois modes différents. Le premier, *step*, fait évoluer le *stash* et donc tous les états associés d'une étape, c'est-à-dire d'un basic-block. L'unité d'exécution est le basic-block (*ce qui lui est reproché comme un défaut par Qsym cf. Section 13.4*). Le deuxième mode, *run*, exécute le programme jusqu'à ce qu'aucun état ne puisse évoluer (*plus aucun état actif*). Le dernier *explore*, laisse l'exécution avancer jusqu'à ce qu'un état ou une adresse spécifiée à l'avance soit atteint.

`angr` fournit une pléthore de stratégies de couverture de programme issues pour la plupart de l'état de l'art ou de papiers de recherche. Ce nombre se justifie par la facilité de personnaliser le comportement de l'exploration du “simulation manager”. Les stratégies intégrées sont : `BFS` (par défaut), `DFS`, `Explorer`, `LengthLimiter`, `LoopSeer`, `ManualMergepoint`, `MemoryWatcher`, `Oppologist`, `Spiller`, `Threading`, `Tracer` et `Veritesting` [16]. Cette dernière est notamment utilisée par `Mayhem` (cf. Section 7.2) et permet la fusion d'états (cf. Section 6.7.5). Une description de chacune des stratégies est fournie dans la documentation en ligne<sup>97</sup>.

Une fois la stratégie d'exploration déterminée et l'exécution d'un pas *step* déclenché par le mode d'exécution, `angr` fait appel à des *engines* (sous classe de `SimEngine`) pour appliquer les effets de bords de l'émulation du basic-block. En fonction du statut de l'état courant exécuté, `angr` cherche successivement parmi les moteurs (*engines*) suivants :

- `SimEngineFailure`, activé si l'état est dans un cas d'arrêt ;
- `SimEngineSyscall`, activé si l'état fait un appel système ;
- `SimEngineHook` activé si l'adresse est “hookée” par l'utilisateur ;
- `SimEngineUnicorn` activé si l'option `unicorn` est activée et qu'aucune donnée symbolique n'est manipulée par l'état ;
- `SimEngineVEX` activé en dernier lieu effectue l'exécution symbolique à proprement parler sur le VEX.

#### 7.5.5 Modélisation des fonctions

`angr` utilise différentes techniques afin de rendre l'exécution symbolique plus malléable. L'une de ces techniques est le “résumé de fonctions”, qui permet de remplacer les fonctions de certaines bibliothèques à l'aide de modèles en Python (appelées `SimProcedures`).

---

97. <https://docs.angr.io/core-concepts/pathgroups#exploration-techniques>

Ces *stubs* ont vocation à éviter la problématique d’explosion des chemins qui peut être introduite en tentant d’exécuter les fonctions de la `libc` et surtout à modéliser la fonction symbolique de manière plus simple. Des modèles de fonctions sont définis pour la `libc`, le noyau et le *loader* Linux, mais aussi pour Windows avec des modèles pour `msvcr`, `ntdll` ou encore `win32`. Enfin, plusieurs modèles pour Java sont aussi proposés (*dans un contexte d’exécution symbolique de Java*). Tous confondus, environ 250 fonctions et appels système sont modélisés. Les modèles sont rigoureusement bien ordonnés dans les sources<sup>98</sup>.

### 7.5.6 Modélisation de la mémoire

La modélisation de la mémoire est inspirée de `Mayhem` [64] où les lectures symboliques sont autorisées, mais les écritures symboliques sont concrétisées. La résolution des adresses est définie par une stratégie de concrétilisation (cf. Section 6.5) héritant de la classe `SimConcretizationStrategy`. Le moteur utilise deux instances de cette classe pour respectivement les lectures et les écritures. Par défaut, `angr` concrétise une écriture si le cardinal des valeurs du pointeur dépasse 128. Pour changer ce comportement, il suffit de remplacer la stratégie d’écriture avec la classe `SimConcretizationStrategyRange` à laquelle il faut fournir l’intervalle maximum de valeurs possibles. L’architecture du framework permet donc de définir n’importe quelle stratégie de concrétilisation.

### 7.5.7 Symbion : Fusion de l’exécution concrète et de l’exécution symbolique

`angr` dépend d’un modèle du système et des bibliothèques sur lesquels il opère afin de rendre l’exécution symbolique possible. Cependant, la modélisation de certains aspects est irréaliste, ou demande une charge de travail bien trop importante. Afin de résoudre cette problématique, la fonctionnalité `Symbion` a récemment été introduite<sup>99</sup> (16 novembre 2018) permettant de combiner l’analyse symbolique d’un programme à une analyse concrète (le rendant au passage un vrai exécuteur concolique). Cela permet à un programme d’être exécuté concrètement d’un point A à un point B (qui pourraient être complexes à modéliser pour `angr`), puis d’exécuter la suite du programme à l’aide d’`angr` et d’un contexte symbolique, en se basant sur l’état concret atteint au point B.

`Symbion` apporte la notion de `ConcretTarget`, une abstraction générique pour tous les environnements qui définit la mémoire, les registres et le contrôle de l’exécution. La transition du mode concret au mode symbolique ne nécessite aucune copie mémoire. En effet, durant la synchronisation de l’état concret, `angr` modifie le contexte de `SimState` afin qu’il soit redirigé vers l’état de la mémoire concrète. Ce mécanisme permet de lire/écrire et modifier dynamiquement la mémoire du programme instrumenté, ce qui rend l’analyse particulièrement polyvalente.

---

98. <https://github.com/angr/angr/tree/master/angr/procedures>

99. [http://angr.io/blog/angr\\_symbion](http://angr.io/blog/angr_symbion)

Rediriger le contexte mémoire de l'état symbolique pour le faire pointer sur l'état concret permet de lire aisément des données en mémoire. Cependant, les écritures réalisées par le moteur d'exécution symbolique ne sont pas appliquées, et restent symbolisées dans le `SimState`. Une concrétisation de ces valeurs apportera une modification effective à l'espace mémoire du programme concret.

Pour le moment, seul GDB semble être supporté comme instrumentation concrète via l'interface de `Avatar2` [258], mais d'autres comme `Pin` [241] ou `DynamoRIO` [48] verront certainement le jour.

#### Interface unifiée `Avatar2`

`Avatar2` se présente comme une interface unifiée pour l'instrumentation concrète de programme. Le but est de fournir une API d'interaction (lecture/écriture mémoire, accès registres etc), indépendant du *backend* utilisé. Cet outil supporte GDB, `OpenOCD`<sup>a</sup>, `QEMU` [26] ou encore `PANDA` [110].

a. <http://openocd.org>

### 7.5.8 Résolution de formules

La résolution de formules s'appuie sur `claripy` fourni par une interface unifiée indépendante du solveur (*très semblable à celle de Z3*). Les solveurs supportés sont Z3 [257], CVC4 [102], ABC [47], mais aussi Z3-str [395] pour la théorie des strings. Ce module est naturellement capable d'exporter les formules vers le SMTLIB2.

### 7.5.9 Utilisation

`angr` propose une API simple d'utilisation via le `SimulationManager`, permettant d'effectuer une exécution concolique. Il propose notamment plusieurs options afin d'initialiser le contexte (par exemple, démarrer l'exécution symbolique dès le point d'entrée, ou plus loin dans le code). Pour les benchmarks de bombes logiques, le choix a été fait de démarrer directement depuis la fonction d'intérêt, à savoir `entry()`. L'initialisation se fait alors grâce à `call_state`<sup>100</sup>, qui reçoit en paramètre l'adresse et les paramètres de la fonction à exécuter. (Dans le cas des `bombes logiques`, le seul paramètre est une chaîne de caractères que l'on déclare comme variable symbolique). Il suffit ensuite de procéder à l'exécution grâce à la méthode `run()` qui exécutera le programme symboliquement jusqu'à ce que tous les états possibles soient épuisés. Il suffit alors de vérifier que l'état final dispose bien de la valeur 1 dans le registre `eax` afin de déterminer si la bombe a été déclenchée. Le code 19 ci-dessous montre le script utilisé pour tester les bombes logiques et la simplicité d'utilisation de `angr`.

100. <https://docs.angr.io/core-concepts/states#state-presets>

---

**Listing 19** Script d'exécution des tests atomiques dans angr

---

```
1 def solve(filename, input_length=128):
2     proj = angr.Project(filename)
3
4     func_addr = proj.loader.find_symbol('entry').rebased_addr
5     func_arg = claripy.BVS("input_string", input_length * 8)
6
7     start_state = proj.factory.call_state(func_addr, func_arg)
8
9     simgr = proj.factory.simulation_manager(start_state)
10
11    simgr.run()
12
13    results = []
14    for state in simgr.deadended:
15        eax = state.solver.eval(state.regs.rax, cast_to=int) & 0xffffffff
16        if eax == 0x1:
17            results.append(state.solver.eval(func_arg, cast_to=bytes))
18
19    return results
```

---

### 7.5.10 Conclusion

`angr` propose une API simple d'utilisation, une bonne documentation<sup>101</sup> et un nombre significatif d'exemples<sup>102</sup> permettant d'apprendre à se servir de l'outil. Il supporte plusieurs formats de fichiers et architectures, est facilement personnalisable et extensible. `angr` est l'un des outils d'exécution symbolique les plus complets, ce qui en fait sa force, mais aussi sa faiblesse.

En effet, `angr` est un projet complexe, basé sur des dépendances internes, mais aussi externes (`capstone`, `unicorn`). Malgré sa complexité, `angr` a néanmoins fait ses preuves lors de la compétition du CGC dans laquelle le `CRS` associé a terminé à la 3<sup>ème</sup> position.

Bien qu'il soit le terrain d'expérimentations pour beaucoup d'analyses et de fonctionnalités, le code est très bien architecturé et relativement facile à prendre en main. Il se prête donc parfaitement à une utilisation dans le cadre d'une combinaison.

---

101. <https://docs.angr.io>

102. <https://github.com/angr/angr-doc/tree/master/examples>

## 7.6 DSE #5 : Triton

L'outil **Triton**<sup>103</sup> a été sélectionné pour la raison évidente qu'il est développé en interne à Quarkslab, mais aussi car il se présente sous la forme d'une bibliothèque. Le premier *commit* date de Mai 2015 et le projet a officiellement été publié sous licence Apache-2 le 3 Juin 2015 lors de la conférence SSTIC<sup>104</sup>. Depuis, il est activement développé, très majoritairement par Jonathan Salwan et totalise plus de 2700 commits. La dernière release est la version 0.6, mais la version utilisée dans cette étude est la future 0.7 qui apportera officiellement le support de l'architecture `aarch64`.

Depuis sa publication, **Triton** a fait l'objet de plusieurs contributions, il a été intégré dans divers outils comme **Radare2**<sup>105</sup>, **IDA Pro** avec le plugin Ponce<sup>106</sup> ou encore **GDB**<sup>107</sup>. Par ailleurs, il a récemment fait l'objet d'un chapitre du livre *Practical Binary Analysis* [9]. Les choix de design et d'implémentation de **Triton** sont portés par les préceptes suivants :

*Poussé par le constat que chaque problématique d'analyse de binaire est différente et requiert une solution différente, Triton se veut rapide et modulaire afin de s'adapter à chaque cas d'usage en fournissant les primitives et directives essentielles pour résoudre le problème rencontré.*

De cette manière, **Triton** propose des directives facilitant l'écriture d'outils d'analyse (plutôt que de fournir un outil clef en main). L'expérience montre qu'un seul outil est trop limité à l'utilisation de celui-ci dans un contexte bien précis et n'offre que très peu de latitude sur des cas d'usages très particuliers. Par exemple, en rétro-ingénierie les programmes à analyser sont souvent uniques et les objectifs très différents. C'est donc naturellement que le choix de conception de **Triton** s'est focalisé sur la mise en place d'une série de primitives dont la granularité d'analyse est celle d'une seule instruction et non d'un programme dans sa globalité. Ce niveau de granularité permet l'usage de **Triton** dans n'importe quel scénario et en combinaison avec n'importe quel autre outil permettant l'accès aux instructions assemblées (ex. **GDB**, **BinaryNinja**, **IDA**, **Radare**, **Pin** ...). En outre **Triton** fournit les primitives d'analyse suivantes :

- émulation concrète
- analyse de teinte dynamique

---

103. <https://triton.quarkslab.com>

104. [https://www.sstic.org/2015/presentation/triton\\_dynamic\\_symbolic\\_execution\\_and\\_runtime\\_analysis/](https://www.sstic.org/2015/presentation/triton_dynamic_symbolic_execution_and_runtime_analysis/)

105. <https://github.com/kamou/pimp>

106. <https://github.com/illera88/Ponce>

107. [https://hitcon.org/2017/CMT/slides/d2\\_s1\\_r0.pdf](https://hitcon.org/2017/CMT/slides/d2_s1_r0.pdf)



FIGURE 7.6 – Logo Triton

- exécution symbolique dynamique
- slicing d'expressions symboliques en arrière

### 7.6.1 Architecture et fonctionnement

#### État interne

Le fonctionnement de **Triton** peut être vu comme une bibliothèque permettant de gérer un état interne, auquel sont envoyés des instructions, qui sont exécutées, tenant à jour cet état avec les effets de bord. Rien n'oblige que les instructions soient séquentielles ou issues d'un basic block. La logique d'alimentation de l'état interne avec des instructions est dépendante du cadre d'utilisation et donc à la charge du développeur.

Plus formellement l'état interne noté  $\Sigma$  peut être défini comme le n-uplet  $\Sigma = (\Sigma_C, \Sigma_S, \Sigma_T)$ , avec  $\Sigma_C$  l'état concret interne conservant les valeurs constantes courantes des registres et cellules mémoires,  $\Sigma_S$  conservant l'état symbolique (valeurs symboliques) et  $\Sigma_T$  conservant l'état de la teinte (booléen) en tout point du programme. L'exécution d'une instruction donne lieu à la mise à jour de chacun des sous-états internes en accord avec la sémantique de l'instruction.

#### Sémantique des instructions

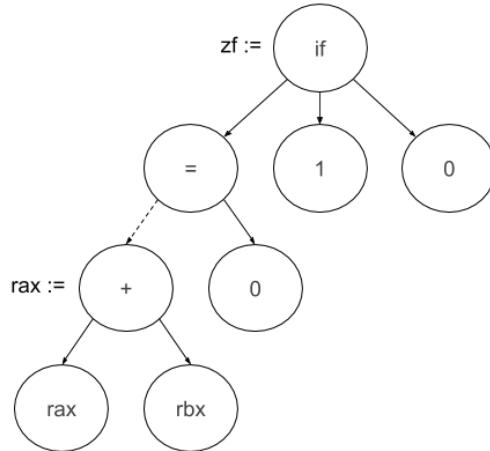


FIGURE 7.7 – AST affecté au registre `rax` et au drapeau ZF après exécution de l'instruction `add rax, rbx`

Le comportement sémantique d'une instruction est modélisé sous la forme d'expressions symboliques. Cette représentation des expressions se matérialise sous la forme d'un arbre représentant l'**AST** de l'instruction (dont les noeuds sont des opérateurs et les feuilles des opérandes). Afin d'illustrer la structure des arbres, par exemple, après exécution de l'instruction x86-64 `add rax, rbx` le registre `rax` et le drapeau `ZF` seront affectés par

l'**AST** de la Figure 7.7. La figure se lit comme suit : *affecte 1 à ZF si rax + rbx vaut 0 sinon affecte 0 à ZF*. Un **AST** sera également créé et assigné aux autres drapeaux : AF, CF, OF, PF et SF. Dans la Figure 7.7, la flèche en pointillés désigne une référence à un AST créé précédemment (partage d'arbre). Cette optimisation permet un partage maximal des arbres d'expressions symboliques. Pour cela, l'algorithme met d'abord à jour les valeurs de registre (**rax** ici) puis met ensuite à jour les nouvelles valeurs de drapeaux en utilisant les références de l'AST de l'expression (référence vers **rax** dans ce cas).

## Exécution

Une fois les arbres de l'instruction courante construits, ils sont interprétés afin d'appliquer la sémantique de l'instruction sur l'état interne  $\Sigma$ . Plus précisément les trois actions suivantes sont effectuées :

- mise à jour de l'état concret  $\Sigma_C$  par simulation pure ;
- mise à jour de l'état symbolique  $\Sigma_S$  (mise à jour des arbres d'expressions symboliques) ;
- mise à jour de l'état de teinte  $\Sigma_T$  tel qu'encodé dans la sémantique de l'instruction.

Ces étapes d'interprétation et de calcul des états sont équivalentes à une simulation de l'exécution d'une instruction et pour chaque exécution ces trois étapes sont appliquées successivement. À noter que les états peuvent interagir entre eux. Ainsi, le composant en charge de l'exécution symbolique a la possibilité d'accéder aux informations concrètes et de teinte afin d'alléger son prédictat de chemin dans le cas d'une application de politique de concrétisation (cf. Section 6.4.2).

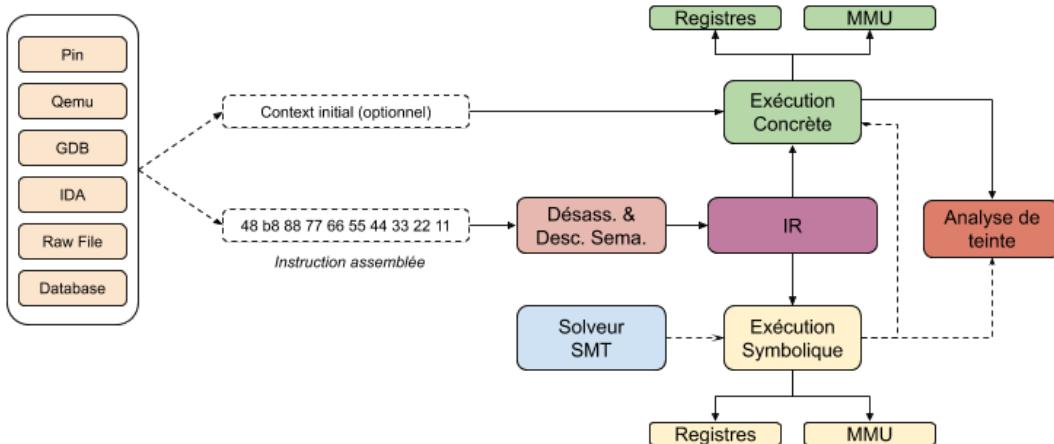


FIGURE 7.8 – Vue globale de la bibliothèque Triton

La Figure 7.8 illustre l'interaction des différents composants de la bibliothèque **Triton**. Dans le processus d'exécution, l'utilisateur commence par définir un contexte concret initial  $\Sigma_C^0$  (ex. état des registres et de la mémoire). Si aucun contexte n'est donné, **Triton** initialise un contexte interne par défaut (avec les valeurs zéro). Ensuite, l'utilisateur fournit l'instruction à exécuter (sous la forme assemblee, ex : 48 b8 88 ...) qui est désassemblée

afin d'obtenir sa mnémonique et ses opérandes<sup>108</sup>. Une fois le type de l'instruction connu, **Triton** représente l'instruction sous la forme d'arbres (cf. 7.6.1) encodant son comportement sémantique sur les registres et la mémoire. Cette description de la sémantique s'appuie sur le manuel technique associé à l'architecture considérée (ex. "Intel 64 and IA-32 Architectures Software Developer's Manual" [186] pour l'architecture x86-64). Actuellement, **Triton** supporte une partie des spécifications des architectures i686, x86-64 et AArch64 (ARM v8.5a).

Lors de l'exécution d'une instruction *inst*, **Triton** prend l'état interne  $\Sigma$  courant et le met à jour en accord avec la sémantique de la nouvelle instruction exécutée ( $\Sigma^i \xrightarrow{\text{exec(inst)}} \Sigma^{i+1}$ ), et ainsi de suite. L'exécution se reposant sur des arbres (qui est une forme de représentation intermédiaire), les analyses (évaluation concrète, symbolique et de teinte) restent génériques vis-à-vis des arcs des différentes architectures supportées (Intel et ARM). Dans le cas du rajout d'une architecture, seul l'encodage des instructions vers la représentation intermédiaire est nécessaire.

## 7.6.2 Résolution des formules et gestion de la mémoire

**Triton** représente ses arbres sous deux formes syntaxiques : en Python et en SMTLIB2 (cf. 6.8). La représentation en SMTLIB2 permet l'usage de n'importe quel solveur supportant ce format. Bien que l'usage de n'importe quel solveur SMT est possible, **Triton** offre une interface native avec Z3 pour la résolution des contraintes. Il utilise la logique QF\_BV car il utilise seulement les bitvecteurs. Les adresses des accès mémoires sont systématiquement concrétisées. Il n'y a donc pas lieu d'utiliser la théorie des tableaux (cf. 6.8.3). Des travaux sont néanmoins en cours pour ajouter le support de cette théorie et pouvoir ainsi raisonner symboliquement sur les pointeurs<sup>109</sup>.

## 7.6.3 Propagation de la teinte

Comme mentionné en 7.6.1, l'exécution d'une instruction donne lieu à la mise à jour d'un état interne de teinte  $\Sigma_T$ . En plus, des effets de bords symboliques, la sémantique de chaque instruction décrit comment est propagée la teinte. Cette teinte est ensuite gérée par le *taint engine* situé dans le fichier *src/libtriton/engines/taint/taintEngine.cpp*. La teinte appliquée est binaire et sa granularité est au niveau de l'objet (registre ou cellule mémoire). Des états plus complexes ou avec une granularité plus fine auraient pu être implémentés, mais le gain en précision ne justifierait pas la perte en performance.

## 7.6.4 Couplage avec une instrumentation dynamique

En marge de la bibliothèque principale, **Triton** fournit un pintool (programme pour Pin [241]) permettant de générer des traces d'exécution symbolique et d'interagir directe-

108. `capstone` est utilisé pour le désassemblage

109. <https://github.com/JonathanSalwan/Triton/pull/723>

ment avec l'exécution concrète de `Pin`. Cela permet notamment de fournir l'état concret initial nécessaire à l'exécution symbolique et de suivre une exécution fournie par un programme tiers. Le support de `Pin` est historiquement dû au fait que `Triton` était lié à `Pin`, mais cela n'est plus le cas depuis la version 0.3. Désormais `Triton` est bibliothèque à part entière ne reposant plus sur une `Dynamic Binary Instrumentation (DBI)` en particulier. Son architecture lui permet donc de se greffer avec différentes `DBI` comme le montre la preuve de concept avec `QBDI`<sup>110</sup>.

### 7.6.5 Stratégies de couverture

Comme présenté en Section 7.6.1, `Triton` a d'abord vocation à exécuter des instructions sur un état interne  $\Sigma$ . La couverture d'un programme, spécifique au cas d'utilisation, est donc déléguée à la charge de l'utilisateur. Pour l'exécution des tests atomiques, un outil (nommé `tritondse`) a tout particulièrement été développé pour effectuer une exploration symbolique des chemins d'un programme.

L'outil en question utilise la bibliothèque `LIEF`<sup>111</sup> pour récupérer les segments du programme ciblé, puis les charge dans l'état interne  $\Sigma_C^0$  de `Triton`. Le script implémente également le comportement de certaines fonctions externes telles que `__libc_start_main`, `atoi`, `malloc`, `free`, `strcpy`... ainsi que certains syscalls tels que `sys_close`, `sys_open`... puis effectue une exploration symbolique des chemins depuis l'*entry point* du programme.

Le principe est d'effectuer une première exécution qui produit un prédicat de chemin. En se basant sur ce premier prédicat de chemin, puis d'inverser des conditions de branchements afin de générer des modèles permettant d'emprunter de nouvelles branches. Ces nouvelles branches sont ensuite exécutées ce qui produit de nouveaux prédicats de chemin et l'opération est répétée jusqu'à ce que toutes les branches soient couvertes.

La Figure 7.9 illustre ce concept d'exploration de chemins. Dans un premier temps sont définies une liste de prédicats à satisfaire (*WL*) et une liste de prédicats empruntés (*DL*). La première exécution est établie depuis un modèle aléatoire afin de fournir une première trace. Cette première trace fournit un prédicat de chemin emprunté ( $\phi_1 = \pi_1 \wedge \pi_2$ ). Ce prédicat est placé dans la liste *DL*. L'algorithme établit ensuite une liste de nouveaux prédicats de chemin n'étant pas dans *DL* à partir du dernier prédicat emprunté ( $\phi_1$ ) et les place dans la liste *WL* (soit :  $\pi_1 \wedge \neg\pi_2$  et  $\neg\pi_1$ ). L'opération est répétée tant qu'il y a des prédicats à satisfaire dans *WL*.

### 7.6.6 Utilisation

La bibliothèque `Triton` fournit un accès à tous ses composants depuis une API accessible en C++ et en Python. Le choix du C++ a été adopté pour des aspects de performances et Python pour le côté "facile" d'intégration et d'utilisation dans une communauté de la sécurité informatique où la présence d'outils en Python est fortement établie.

---

110. [http://shell-storm.org/repo/Notepad/qbdi\\_with\\_triton.txt](http://shell-storm.org/repo/Notepad/qbdi_with_triton.txt)

111. <https://lief.quarkslab.com/>

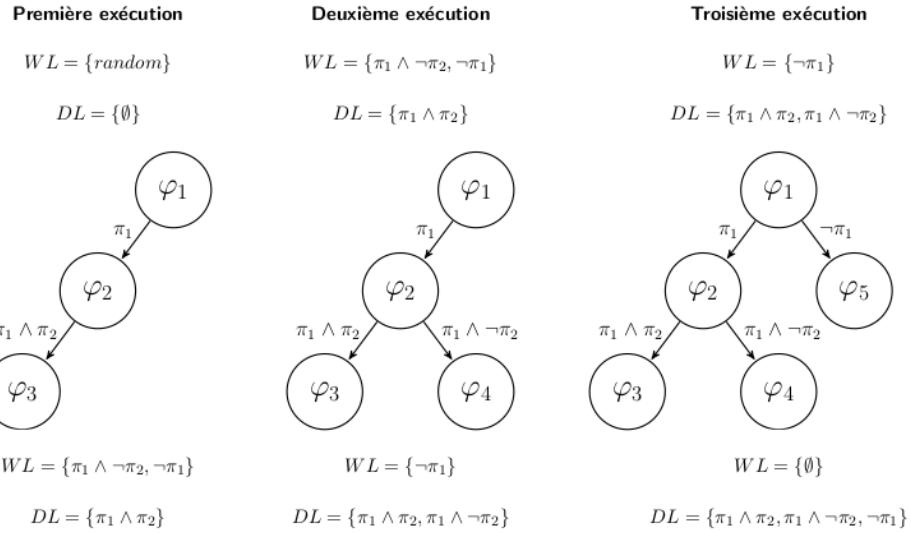


FIGURE 7.9 – Exemple d’exploration symbolique de chemins

Comme décrit en Section 7.6.1, Triton reçoit uniquement les instructions à exécuter, il n'est pas responsable de l'ordonnancement de l'exécution ni de la gestion des appels externes tels que les *syscall* ou les fonctions externes (ex. *atoi*, *malloc*, ...). C'est donc à la charge de l'utilisateur de simuler ces comportements. Par exemple, le code illustré par le Listing 20 simule une exécution. En partant du principe que tout le code du programme a été chargé dans la mémoire interne de Triton, le script récupère le code pointé par le pointeur d'instruction *program counter* (ligne 3), puis crée une nouvelle **Instruction** (ligne 6) ensuite exécutée par Triton (ligne 10). Cette exécution met à jour le *program counter* qui sert ensuite à récupérer la prochaine instruction (ligne 18), et ainsi de suite.

### 7.6.7 Gestion des appels de bibliothèques

Comme mentionné ci-dessus Triton ne gère pas les appels externes. L'exemple du Listing 21 illustre la simulation d'un appel externe (ici *strcpy*). Le code commence par récupérer les arguments depuis les états internes de Triton (lignes 5 et 6) et propage les expressions symboliques de la source vers la destination (lignes 8 à 13).

Ces deux exemples montrent à la fois les contraintes qu'implique une granularité de l'ordre de l'instruction (et non d'un programme comme le fait **Manticore** et **angr**), mais montrent également la souplesse que cela peut apporter. À noter qu'une fois les routines créées (ex. *malloc*, *atoi*, ...), ces dernières peuvent ensuite être réutilisées dans des projets ultérieurs sans grande modification.

**Listing 20** Exemple pour simuler une exécution avec Triton

---

```
1 def emulate(self, pc):
2     # Fetch opcodes
3     opcodes = self.ctx.getConcreteMemoryAreaValue(pc, 16)
4
5     # Create the Triton instruction
6     instruction = Instruction(opcodes)
7     instruction.setAddress(pc)
8
9     # Execute the instruction
10    if self.ctx.processing(instruction) == False:
11        self.debug('[-] Instruction not supported: %s' %(str(instruction)))
12        break
13
14    # Print the instruction
15    print(instruction)
16
17    # Next pc
18    pc = self.ctx.getConcreteRegisterValue(self.getPcRegister())
19
20    return
```

**Listing 21** Exemple pour simuler un appel externe avec Triton

---

```
1 # Simulate the strcpy() function
2 def strcpyHandler(self):
3     self.debug('[+] strcpy hooked')
4
5     dst = self.ctx.getConcreteRegisterValue(self.getArgumentRegister(0))
6     src = self.ctx.getConcreteRegisterValue(self.getArgumentRegister(1))
7     for index in range(len(self.getMemoryString(src))):
8         dmem = MemoryAccess(dst + index, 1)
9         smem = MemoryAccess(src + index, 1)
10        cell = self.ctx.getMemoryAst(smem)
11        expr = self.ctx.newSymbolicExpression(cell, "strcpy byte")
12        self.ctx.setConcreteMemoryValue(dmem, cell.evaluate())
13        self.ctx.assignSymbolicExpressionToMemory(expr, dmem)
14
15    return (Enums.CONCRETIZE, dst)
```

### 7.6.8 Conclusion

L'approche de **Triton** est la mise à disposition d'une bibliothèque permettant la création d'outils d'analyse plutôt que des outils comme `angr` [320] ou `Manticore` [33] fournis clés en main. Il en est même l'antithèse de ces outils qui visent à fournir autant de fonctionnalités que possible, lui se veut simple modulaire et performant. Des expérimentations internes montrent qu'il concurrence des émulateurs comme `unicorn` en émulation pure (alors qu'il transforme quand même les expressions en AST). Cette modularité, le rend utilisable dans n'importe quel scénario et dans n'importe quel outil, bien que cela nécessite de connaître la cible et de développer ses propres outils. Ce choix de conception peut sembler un peu archaïque en comparaison des nouveaux outils, mais cela apporte beaucoup de latitude quant à l'usage qu'il est possible d'en faire. **Triton** se veut simple et stable, même s'il propose peu de fonctionnalités, celles-ci passent à l'échelle, ce qui reste le point faible du DSE et des outils existants.



## Évaluation des outils

---

### 8.1 Bombes logiques : Évaluation & Analyse des résultats

La suite de test, conçue avant tout pour tester les spécificités des outils de DSE, a permis d’appréhender le comportement de ceux sélectionnés sur un éventail de 70 tests. Le tableau 8.1 donne les résultats de chaque outil sur chacun des tests. Les résultats par unité de test sont détaillés dans les sections ci-dessous. Pour rappel, ces résultats sont à interpréter au vu de la connaissance acquise sur les outils. Il serait très certainement possible d’optimiser les résultats avec une meilleure compréhension des options et du fonctionnement interne. Par ailleurs, il est à noter que les tests ARM de `angr` et `Manticore` ont été exécutés sur le serveur x86 en raison de difficultés d’installation sur le serveur ARM (*notamment `unicorn` pour `angr`*). La nature purement symbolique de ces deux outils fait que cela n’avait aucun impact sur l’exécution. De plus `Manticore` ne supporte que l’ARM 32 bits (ARMv7), les benchmarks ont donc été faits sur ARMv7. En sus, comme mentionné en Section (cf. 7.3.6) la suite a légèrement été modifiée pour `Manticore` afin de compenser le manque de support de `fgets` (remplacée par `read`). Enfin, la suite pour `Triton` a été compilée avec `CMD_LINE_INPUT` pour recevoir les arguments sur `argv` plutôt que `stdin`.

#### 8.1.1 Résultats de UT\_1

Tous les outils arrivent à résoudre les tests de tableaux à base d’index constants (CE\_1). Les choses se dégradent pour la propagation CE\_2 via des fichiers ou la pile. Les deux explications possibles sont soit le manque de support de l’appel système ou de bibliothèque, soit plus plausiblement la perte de teinte sur laquelle les outils se basent. Les résultats en termes de lecture/écriture symbolique en mémoire sont très modestes. En effet, au-delà d’une lecture ou d’une écriture symbolique, les outils ne sont plus capables de raisonnement. Ainsi, un double déréférencement symbolique de pointeur dans une structure est hors de portée des outils. À noter qu’il est difficile d’expliquer la disparité

## Chapitre 8. Évaluation des outils

	Id	Nom	angr 8.18		Manticore 0.2.5		KLEE 2.1		Triton 0.7	
			x64	ARMv8	x64	ARMv7	Annot.	∅	x64	ARMv8
UT_1	CE_1	CId_1 array_sample1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_2	CId_2 array_sample2	✓	✓	✓	✓	✓	✓	✓	✓
	CE_3	CId_3 array_sample3	✓	✓	✓	✓	✓	✓	✓	✓
	CE_4	CId_4 array_sample4	✓	✓	✓	✓	✓	✓	✓	✓
	CE_5	CId_5 array_sample5	✓	✓	✓	✓	✓	✓	✓	✓
	CE_6	CId_6 array_sample6	✓	✓	✓	✓	✓	✓	✓	✓
	CE_7	CId_7 file_cp_11	X*	X	X	✓	X‡	X‡	X‡	X‡
	CE_8	CId_8 file_posix_cp_11	✓	✓	X	X	X‡	X‡	X‡	X‡
	CE_9	CId_9 stack_cp_11	✓	✓	✓	✓	X§	X§	✓	✓
UT_2	CE_1	CId_10 stackarray_sm_11	✓	✓	✓	X	✓	✓	✓	✓
	CE_2	CId_11 stackarray_sm_12	X*	X*	✓	X	✓	✓	✓	✓
	CE_3	CId_12 stackarray_sm_ln	X†	X†	X†	X	✓	✓	✓	✓
	CE_4	CId_13 pointers_sj_11	X*	X*	✓	✓	✓	✓	✓	✓
	CE_5	CId_14 stackarray_sm_store_11	X*	X*	✓	X	✓	✓	✓	X†
	CE_6	CId_15 stackarray_sm_store_12	✓	X*	✓	X	X*	X*	✓	X†
	CE_7	CId_16 stackarray_sm_rw_11	X*	✓	✓	X	✓	✓	X†	X†
	CE_8	CId_17 atoi_ef_12	✓	✓	✓	X†	✓	✓	✓	✓
	CE_9	CId_18 pow_ef_12	✓	X*	X	X*	X	X	X	X
UT_3	CE_1	CId_19 printint_int_11	✓	✓	✓	✓	✓	✓	✓	✓
	CE_2	CId_20 rand_ef_12	X*	X*	X†	X	X†	X†	X†	X†
	CE_3	CId_21 file_csv	X*	X*	X*	X*	✓	✓	✓	✓
	CE_4	CId_22 file_posix_csv	X*	X*	X†	X†	✓	✓	X‡	X‡
	CE_5	CId_23 pid_csv	X*	X*	X†	X†	X	✓	X‡	X‡
	CE_6	CId_24 malloc_1	✓	✓	✓	✓	✓	✓	X‡	X‡
	CE_7	CId_25 memset_1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_8	CId_26 syscall_alarm	X	X*	X‡	✓	X§	X§	X‡	X‡
	CE_9	CId_27 syscalls_open_close	X	X*	X†	X†*	X§	✓	✓	X‡
UT_4	CE_1	CId_28 syscalls_read_write	✓	X*	X	X	X§	X§	X‡	X‡
	CE_2	CId_29 syscall_time	✓†	✓	✓	X†	✓	✓	X‡	X‡
	CE_3	CId_30 pmccntr	✓†	X*	✓	✓	X§	✓	X‡	X‡
	CE_4	CId_31 cpuid_1	X	X*	X	/	X§	X	X‡	X‡
	CE_5	CId_32 cpuid_2	✓†	X*	X†	/	X§	X	X‡	X‡
	CE_6	CId_33 string_sample1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_7	CId_34 string_sample2	✓	✓	X†	✓	✓	✓	✓	✓
	CE_8	CId_35 string_sample3	✓	✓	✓	X	✓	✓	✓	✓
	CE_9	CId_36 string_sample4	X†	X†	✓	X†	✓	✓	✓	✓
UT_5	CE_1	CId_37 propagation_sample3	✓	✓	X†	X†	✓	✓	✓	✓
	CE_2	CId_38 float_simple1	✓	✓	X	X*	X	X	X	X
	CE_3	CId_39 atof_ef_12	X†	X*	X†	X†	X†	X†	X	X
	CE_4	CId_41 sin_ef_12	X†	X*	X	X†	✓	✓	X	X
	CE_5	CId_42 ln_ef_12	X	X*	X	X*	X	X	X	X
	CE_6	CId_43 5n+1_lo_11	X†	X†	X†	X†	✓	✓	✓	X†
	CE_7	CId_44 collaz_lo_11	X†	X†	X†	X†	✓	✓	✓	X†
	CE_8	CId_45 loop_sample1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_9	CId_46 loop_sample2	X†	X†	✓	✓	✓	✓	✓	✓
UT_6	CE_1	CId_47 loop_sample3	X†	X†	X†	X†	✓	✓	✓	✓
	CE_2	CId_48 df2cf_cp_11	✓	✓	✓	X†	✓	✓	✓	✓
	CE_3	CId_49 sample_path1	✓	✓	X†	X†	✓	✓	✓	✓
	CE_4	CId_50 sample_path2	X*	X*	X†	X†	X†	X†	✓	✓
	CE_5	CId_51 sample_path3	✓	✓	X†	X†	✓	✓	✓	✓
	CE_6	CId_52 sample_path4	X*	X*	X†	X†	X†	X†	✓	✓
	CE_7	CId_53 sample_path5	✓	✓	X†	X†	✓	✓	✓	X*
	CE_8	CId_54 sample_path6	X†	X†	✓	✓	✓	✓	✓	✓
	CE_9	CId_55 bof_sample1	X*	X*	X†	X†	X	X	✓	✓
UT_7	CE_1	CId_56 bof_sample2	X*	X*	X†	X†	X	X	✓	✓
	CE_2	CId_57 stack_bo_11	X*	X*	X†	X†	✓	✓	✓	✓
	CE_3	CId_58 iof_sample1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_4	CId_59 iof_sample2	✓	✓	✓	✓	✓	✓	✓	✓
	CE_5	CId_60 integer_overflow_1	✓	✓	✓	✓	✓	✓	✓	✓
	CE_6	CId_61 off_by_one_1	✓	✓	✓	X†	✓	✓	✓	✓
	CE_7	CId_62 off_by_one_2	✓	✓	✓	X†	✓	✓	✓	✓
	CE_8	CId_62 off_by_one_3	✓	✓	✓	✓	✓	✓	✓	✓
	CE_9	CId_63 stackarray_sm_12_[...]	X*	✓	✓	✓	✓	✓	✓	✓
UT_8	CE_1	CId_64 use_after_free_1	✓	✓	✓	✓	✓	✓	✓	X‡
	CE_2	CId_65 use_after_free_2	✓	✓	✓	✓	X*	✓	X‡	X‡
	CE_3	CId_66 format_string_1	X*	X*	X†	X†	✓	✓	X‡	X‡
	CE_4	CId_67 format_string_2	X*	✓	X†	X†	✓	✓	X‡	X‡
	CE_5	CId_68 format_string_3	X*	X*	X†	X†	✓	✓	X‡	X‡
	CE_6	CId_69 read_anywhere	X†	X*	X†	X†	X	X	X†	X†
	CE_7	CId_70 stackoutofbound_sm_12	X*	✓	✓	✓	X	✓	✓	✓

T : Échec par timeout, † : validé mais entrée générée non rejouable, ‡ : syscall ou fonction non gérés

§ : assemblage inline non supporté, \* : erreur d'exécution quelconque

TABLE 8.1 – Résultats détaillés des tests atomiques de DSE

des résultats sur la mémoire pour **Manticore** en x86\_64 et en ARM. KLEE semble bien gérer les lectures symboliques, mais aussi les écritures symboliques lorsque ce ne sont pas des doubles indirections symboliques (*comme dans stackarray\_sm\_store\_l2*). Bien que **Triton** ne représente pas la mémoire sous la forme d'un tableau symbolique, il est tout de même possible d'effectuer la couverture des états (toutes les valeurs possibles des lecture/écriture), ce qui permet de valider CE\_3 ainsi que CE\_4.

### 8.1.2 Résultats de UT\_2

La modélisation des appels de bibliothèques externes (CE\_1) montre que les plus importants, comme `malloc`, `memset`, `atoi`, sont plutôt bien gérés par **angr**, **Manticore**. Avec **Triton** c'est à l'utilisateur de simuler le comportement de toutes *entités* externes (syscalls, appels externes), il est donc théoriquement possible de résoudre les tests CE\_1 et CE\_2. La validation de certains, comme `pid_csv`, est aléatoire dans la mesure où elle se base sur la valeur de **Process IDentifier (PID)** que beaucoup d'outils comme KLEE concrétisent. Les appels systèmes sont moins bien gérés, et ce même avec un mécanisme de Linux symbolique comme dans **Manticore** (cf. 7.3.4). KLEE n'a pas été en mesure de les résoudre, car les syscalls sont codés en assembleur inline. La modélisation des instructions non déterministes montre que RDTSC est bien supporté (*pour la plupart avec un compteur incrémenté à chaque instruction*) mais que `cpuid` utilise plutôt des valeurs hardcodées. C'est le cas de **Triton** et de **Manticore** qui dans *libtriton/arch/x86/x86Semantics.cpp* et *manticore/native/cpu/x86.py* définissent différentes valeurs en dur en fonction de la valeur de `eax` en paramètre.

#### Note

Sur ARM, `pmccntr` est un registre dont la lecture se fait normalement en mode privilégié (EL1). Les outils supportant l'instruction ne tiennent donc pas compte de la notion de priviléges durant leur exécution.

**angr** gère bien mieux les strings que **Manticore** qui semble peiner à les résoudre faute de temps. Cela se justifie bien, mais la gestion des flottants est très faible, quel que soit l'outil.

### 8.1.3 Résultats de UT\_3

Les résultats des tests d'explorations sont très instructifs et montrent les difficultés qu'ont les outils de DSE à couvrir les boucles. Un test avec un timeout de ~10 minutes a validé l'idée que tous les `sample_path` auraient été validé par **Manticore** avec un plus gros timeout. À noter que pour **Manticore**, `sample_path4` renvoie un code de retour symbolique qui est par défaut résolu via un solveur. Or comme il ne semble pas essayer d'itérer les différentes valeurs possibles, pour les tests il a toujours retourné 0 alors qu'il serait en mesure de valider le test. Ces tests éprouvent bien la capacité des DSE à couvrir des

chemins, comme en attestent les 307 884 cas de tests générés par KLEE sur `sample_path6` (cf. Figure du CFG 5.1) en l'espace de 5 minutes.

L'UT\_3 est particulièrement catastrophique pour `angr`. Une seule des bombes de CE1 a été déclenchée, et seules 4 des 7 bombes de CE2 ont été déclenchées. L'exploration du programme par `angr` semble peu efficace. Cependant, plusieurs méthodes d'exploration existent, comme le montre la Section 7.5.4. Certaines d'entre elles pourraient s'avérer plus efficaces, mais n'ont pas été testées. C'est le cas pour `Triton` qui n'a pas eu de peine à découvrir toutes les bombes.

### 8.1.4 Résultats de UT\_4

Les résultats pour les débordements de buffers sont étonnamment timorés et seul KLEE arrive à tous les résoudre. Il en va de même pour le débordement d'entiers et les off-by-one. La détection des Use-After-Free est seulement “gardée” par une condition sur un caractère, le déclenchement est donc aisé. En revanche, les outils semblent ignorer le bug. Par exemple, pour `use_after_free_1`, `Manticore` renvoie TRIGGERED sans plus de warning. Pour `use_after_free_2`, il renvoie le code de retour 127 sur x86 et crash en ARM avec l'erreur `unicorn.unicorn.UcError: Invalid instruction (UC_ERR_INSN_INVALID)`. Encore une fois, le double-free ne semble pas l'alerter. De la même manière pour `read_anywhere`, la lecture à un index symbolique est bien détectée et `Manticore` semble s'arrêter à cette détection. Il n'essaye de pas de tirer profit de cette lecture pour récupérer le flag et renvoyer TRIGGERED. KLEE a pour certains tests un comportement analogue, car il détecte bien toutes les vulnérabilités (cf. message de log ci-dessous 22) et génère des cas de tests en conséquence, mais il n'en tire pas profit pour continuer la couverture.

---

#### Listing 22 Message de détection d'erreurs de KLEE

---

```
1 KLEE: ERROR: use_after_free_1.c:39: memory error: out of bound pointer
2
3 KLEE: ERROR: use_after_free_2.c:32: memory error: invalid pointer: free
4
5 KLEE: ERROR: read_anywhere.c:23: memory error: out of bound pointer
6
7 # bof_sample1 et bof_sample2
8 KLEE: ERROR: libc/string/strcpy.c:27: memory error: out of bound pointer
```

---

Une nouvelle fois, les résultats d'`angr` apparaissent plutôt faibles. En effet, ce dernier n'a déclenché aucune bombe relative aux dépassesments de tampons, ou aux `format strings`. Une étude plus approfondie a été menée pour expliquer les résultats décevants de `angr` pour CE\_1 sur les débordements de tampon. Le problème est notamment dû au script d'exécution (cf. 19) mal adapté pour résoudre ces tests. L'annexe E décrit plus en

détail le problème. En sus, lors de l'exécution des benchmarks, le problème suivant est survenu :

---

```
Exit state has over 257 possible solutions. Likely unconstrained; skipping.
```

---

Celle-ci se produit lorsque le pointeur d'instruction peut avoir plus de 257 valeurs possibles, ce qui est effectivement le cas pour **CE\_1** lorsque les données écrasent la valeur de retour sur la pile. Dans ce cas-là, l'exécution passe l'état dans le mode *unconstrained* et arrête son exploration.

### 8.1.5 Cas de tests

Les résultats des benchmarks ne représentent pas l'aptitude des outils à générer des cas de tests qui varie énormément d'un outil à l'autre. Certains, comme **Manticore**, ont été configurés pour s'arrêter dès qu'une entrée générée renvoyait **TRIGGERED**, et d'autres comme **KLEE**, fonctionnaient jusqu'à la couverture complète des chemins ou timeout. Même avec ce biais **Manticore** (en x86) a généré 287 cas de tests pour 23 timeouts alors que dans le même temps **KLEE** a généré plus d'un million de cas de tests pour 3 timeouts. Cela donne un aperçu des performances et du nombre de chemins générés par seconde. Contre-intuitivement, le benchmark de **KLEE** avec les annotations a généré 1005178 cas de tests pour 731512 sans annotations (différence de 273K). Cela s'explique par le fait qu'étant facilité avec les entrées symboliques, **KLEE** peut passer plus de temps à couvrir un programme dans les 5 minutes d'exécution imparties. À ce titre, le critère d'évaluation pour **UT\_3/CE\_2** (Chemins), joue pleinement son rôle puisque la moitié de tous les chemins générés le sont par ce critère (466586 chemins), avec 307884 pour **sample\_path6** à lui seul. À titre informatif, le tableau 8.2 donne la répartition du code de retour des cas de tests générés par **KLEE**.

Code	Nom	Nombre	%
0	NOT_TRIGGERED	760385	75.6 %
1	TRIGGERED	228228	22.7 %
124	timeout	3591	0.35 %
135	Erreur de bus (bus-error)	69	0.01 %
139	Erreur de segmentation	12904	1.28 %

TABLE 8.2 – Code de retour cas de tests KLEE

Chaque rejet était limité par un timeout, car certains cas de tests génèrent des entiers 32 bits déroulés un par un dans des boucles dans **UT\_3/CE\_1**. Sans être vulnérables, les inputs générant des timeouts sont nonobstant symptomatiques d'entrées pouvant générer des **Denial-of-Service (DOS)**.

## 8.2 LAVA : Évaluation & Analyse des résultats

Les résultats obtenus sont très décevants, quel que soit l'outil. Aucun d'eux n'a été en mesure de parcourir les deux programmes pour en extraire les bugs, et ce malgré leur petite taille ( $\sim 950$  Ko en x86 et 750 Ko en ARM). Cependant, cette vacuité de résultats trouve plusieurs explications selon l'outil. **Manticore** plante avec une erreur vraisemblablement non prévue sur **base64** en x86. Le binaire **uniq** en x86 épouse toute la mémoire et fait crasher la machine ( $> 32$  Go). Cela semble aussi être un bug. Sur ARM, le manque du support de l'appel système **sys\_arm\_fadvise64\_64** interrompt la couverture du programme de manière prématuée. Ce même appel système semble poser problème à **KLEE** qui s'arrête dès le premier chemin. Il n'a pas été possible de creuser l'origine du problème. La consommation mémoire et le support de l'appel système **fadvise64** semble aussi être la cause de l'échec de **angr** (et confirmé par le papier **Qsym** [380]). À l'inverse de **angr** ou **Manticore**, **Triton** est une bibliothèque d'exécution symbolique et non un outil autonome clefs en main. Il est à la charge de l'utilisateur de développer ses propres analyses personnalisées pour les programmes à auditer et le script rudimentaire de couverture utilisé sur les tests atomiques ne s'appliquerait pas sur LAVA-M. Il nécessiterait l'ajout de bien d'autres modèles de fonctions pour être efficace. Ainsi, il n'est pas possible de tester **Triton** sur la suite LAVA-M.

CE	Nom	angr 8.18		Manticore 0.2.5		KLEE 2.1		Triton 0.7	
		x86_64	ARM	x86_64	ARM <sup>112</sup>	Annot.	$\emptyset$	x86_64	ARM
CE_1 : Nombre de bugs trouvés	base64	0/44	0/44	0/44	0/44	1/44	1/44	0/44	0/44
	uniq	0/28	0/28	0/28	0/28	0/22	0/28	0/28	0/28

TABLE 8.3 – Résultats des benchmarks pour UT\_5 : Passage à l'échelle

## 8.3 Synthèse & Conclusion des benchmarks

### 8.3.1 Synthèse des tests atomiques

Les résultats quantitatifs synthétiques des tests atomiques sont donnés sur le tableau 8.4. D'une architecture à l'autre, et d'un outil à l'autre, les résultats sont relativement uniformes pour les DSE fonctionnant au niveau binaire. L'uniformité entre x86 et ARM conforte l'idée qu'à partir du moment où un outil supporte une architecture, les résultats seront similaires d'une architecture à l'autre.

Dans le contexte du projet, un intérêt tout particulier est à porter sur la détection de vulnérabilités de UT\_4. À ce titre, la plupart des outils semblent avoir un support au moins partiel des différentes vulnérabilités. Cependant, la base de bombes logiques existante est rétrospectivement un choix très discutable. En effet, en plus d'être buggés

112. résultats en ARMv7

### 8.3. Synthèse & Conclusion des benchmarks

	Critères	angr 8.18		Manticore 0.2.5		KLEE 2.1		Triton 0.7	
		x86_64	ARM	x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_1	CE_1 : Tableau	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
	CE_2 : Propagation	2/3	2/3	1/3	2/3	0/3	0/3	1/3	1/3
	CE_3 : Lecture mémoire symbolique	1/4	1/4	3/4	1/4	4/4	4/4	4/4	4/4
	CE_4 : Ecriture mémoire symbolique	1/2	0/2	2/2	0/2	1/2	1/2	2/2	0/2
	CE_5 : Lecture/Ecriture mémoire symbolique	0/1	1/1	1/1	0/1	1/1	1/1	0/1	0/1
UT_2	CE_1 : Fonctions externes	5/9	4/9	5/9	4/9	5/9	6/9	5/9	5/9
	CE_2 : Appels système	2/4	1/4	1/4	1/4	1/4	2/4	1/4	0/4
	CE_3 : Instructions non déterministes	2/3	0/3	1/3	1/3	0/3	1/3	0/3	0/3
	CE_4 : Gestion des strings	4/5	4/5	3/5	2/5	5/5	5/5	5/5	5/5
	CE_5 : Nombres flottants	1/4	1/4	0/4	0/4	1/4	1/4	0/4	0/4
UT_3	CE_1 : Boucles	1/5	1/5	2/5	2/5	5/5	5/5	5/5	3/5
	CE_2 : Chemins	4/7	4/7	2/7	2/7	5/7	5/7	7/7	6/7
UT_4	CE_1 : débordement de tampon	0/3	0/3	0/3	2/3	1/3	1/3	3/3	3/3
	CE_2 : débordement d'entiers	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3
	CE_3 : Off-by-one	3/4	4/4	3/4	3/4	4/4	4/4	4/4	4/4
	CE_4 : Use-after-free	2/2	2/2	2/2	1/2	2/2	2/2	0/2	0/2
	CE_5 : Format string	0/3	1/3	0/3	3/3	3/3	3/3	0/3	0/3
	CE_6 : Accès mémoire invalide	0/2	1/2	1/2	1/2	0/2	1/2	1/2	1/2
Total bombes logiques :		37/70	36/70	34/70	34/70	47/70	51/70	47/70	41/70
UT_5	CE_1 : Nombre de bugs trouvés base64	0/44	0/44	0/44	0/44	1/44	1/44	0/44	0/44
	CE_1 : Nombre de bugs trouvés uniq	0/28	0/28	0/28	0/28	0/28	0/28	0/28	0/28
Total :		37/142	36/142	34/142	34/142	48/142	52/142	47/142	41/142

TABLE 8.4 – Résultats synthétiques des benchmarks des outils de DSE

et d'avoir requis de nombreuses modifications, les tests de UT\_4 confondent détection et exploitation des vulnérabilités. La plupart des challenges, pour être résolus, doivent être exploités automatiquement par l'outil, alors que le comportement nominal des outils est de s'arrêter lorsqu'ils rencontrent un débordement ou autre (ils ne sont pas censés exploiter automatiquement le bug pour continuer la couverture “comme si de rien n’était”). À cet égard, la plupart des outils détectent la vulnérabilité même s'ils ne l'exploitent pas. Pour tous les outils, la logique de détection des bugs est plutôt bien maîtrisée et implémentée, mais aucun d'entre eux ne code la logique d'exploitation. À juste titre celle-ci est spécifique au but à atteindre.

De manière générale, les DSE se défendent bien et peinent sur les problèmes connus de combinatoire de chemin, de modélisation et gestion de la mémoire. Malgré les différentes erreurs et bugs rencontrés, les résultats sont très satisfaisants pour PASTIS, et c'est presque un sans-faute pour KLEE, dont les seuls défauts sont de ne pas gérer l'assembleur inliné et certains syscalls. Il a détecté toutes les vulnérabilités, même s'il n'a pas été en mesure de toutes les “exploiter” (*ce qu'aucun outil n'est programmé à faire intentionnellement*). Les résultats légèrement faibles de angr s'expliquent très largement par le manque de connaissances des arcanes de configuration et de paramétrages de l'outil. Le temps alloué à la compréhension de l'outil est donc à prendre en compte dans l'interprétation des résultats. angr est un outil relativement complet, ses résultats ne reflètent pas ses réelles capacités. À l'inverse, Triton est développé à Quarkslab la connaissance de cet outil est donc totale. Ainsi, il est possible d'affirmer que le résultat théorique de **64/70** est atteignable sur les tests atomiques, moyennant l'amélioration du script utilisé pour effectuer

la couverture. Comme les autres outils, la principale limitation de la bibliothèque **Triton** pour cette suite de test est le support des opérations sur les flottants. Il est envisageable d'ajouter le support si cela s'avérait nécessaire. Les résultats sur **AArch64** sont légèrement en deçà des résultats sur **x86\_64**, car le support de cette architecture est donc encore un peu jeune. Celui-ci a été introduit dans **Triton** début Janvier<sup>113</sup> sera officiellement intégré dans la prochaine release (0.7) de l'outil.

### 8.3.2 Synthèse des tests de passage à l'échelle

Ni **angr**, ni **Manticore**, ni **KLEE**, n'ont été en mesure de fournir des résultats sur les benchmarks LAVA-M. Ainsi, les résultats obtenus ne permettent pas de statuer sur l'aptitude des différents outils à passer à l'échelle sur ce type de programmes. Les raisons sont principalement techniques et non pas dûes à des problèmes de passage à l'échelle. En effet, les différents problèmes rencontrés et particulièrement le support de **fadvise64** aussi bien en ARM qu'en x86 a entravé les résultats. Pour obtenir des résultats il aurait été nécessaire de modifier les binaires de LAVA-M pour désactiver l'appel système **fadvise64**, ce qui n'a pas été fait faute de temps. Néanmoins, cela montre l'importance pour un DSE d'être en mesure de modéliser les différents appels systèmes et autre sources de non-determinisme dans le programme. Par ailleurs, sur **Manticore**, aussi bien les tests atomiques que les tests LAVA-M ont soulevé plusieurs bugs qui sont indéniablement non voulu.

### 8.3.3 Synthèse sur les outils de DSE

Parmi les ressources étudiées, **Mayhem** (cf.7.2) bien que présentant une architecture et des caractéristiques très intéressantes, n'a pas pu être testé sur la suite de test élaborée. **Mayhem** donne néanmoins un très bon aperçu d'une architecture de DSE passant à l'échelle et suffisamment performante pour gagner le **CGC**.

L'aptitude des différents DSE à gérer le réseau a été occultée dans la suite de test, mais tous ceux testés, et en particulier **KLEE**, **angr** possèdent un support au moins expérimental du réseau (considéré comme une lecture symbolique dans un fichier).

Enfin, sur ce genre d'outils aussi spécialisés, il convient de considérer que le manque de connaissances sur le fonctionnement ne permet pas de tirer le meilleur profit de chacun des outils. À cet égard, il est certain que les résultats obtenus avec **Triton** ont légèrement été favorisés.

En termes de performance, les résultats sont assez inégaux. **Manticore** additionne à l'exécution symbolique une émulation complète du système particulièrement lent. Au-delà des considérations de rapidité et de résultats quantitatifs aux benchmarks, les deux

---

113. <https://github.com/JonathanSalwan/Triton/commit/349da6be947169ad88d3d14e11e4477\416e0e706>

principaux critères à considérer sont l'utilisation qui va en être faite, et la facilité de prise en main.

**Utilisation** Les deux principales utilisations qui peuvent être faites d'un DSE sont soit un usage *standalone* (*online*) où le DSE effectue de la couverture et de la génération de test. Soit, à la manière de **Mayhem**, un DSE qui résout les prédictats de chemins qui lui sont donnés par un autre outil. À cet égard, l'architecture sous forme de bibliothèque de **Triton** le rend particulièrement adapté à cet usage.

**Adaptabilité** En fonction de l'usage considéré, certains DSE se prêtent plus que d'autres. Pour un usage *standalone*, **Triton** serait limité en fonctionnalités et **KLEE** serait relativement difficile à adapter à d'autres besoins que celui pour lequel il est conçu. À l'inverse, **angr** et **Manticore**, tous deux développés en Python, sont suffisamment faciles à prendre en main pour être adaptés à différents scénarios. Dans un contexte de combinaison, ces deux outils pourraient certainement être adaptés. Néanmoins, par essence, **Triton** est invariablement l'outil le plus facilement adaptable au contexte, étant lui-même développé par Quarkslab.



# **Quatrième partie**

## **Fuzzing**



# Chapitre 9

---

## État de l'art : Fuzzing

---

### 9.1 Introduction

La notion de fuzzing, introduite dans les années 90 par Barton P. Miller [251], a depuis généré un grand nombre de publications et un regain d'intérêt ces dernières années [275]. Ceci est en partie conjecturel avec un besoin de plus en plus important en fuzzing dû à des failles de plus en plus critiques, mais cela est aussi dû à l'arrivée de fuzzers faciles à prendre en main tels que `libfuzzer` [338], `Honggfuzz` [335] et surtout `AFL` [383]. Ce dernier a suscité un engouement sans précédent et s'est vite imposé comme l'outil de référence. De nombreux dérivés et outils se sont bâtis autour de cet outil pour former un écosystème très complet<sup>114</sup>. Avant cela, le fuzzing était initialement défini comme “*programme qui génère un flux de caractères aléatoires consommés par un programme cible*”. La définition a depuis évolué pour enlever le caractère purement aléatoire des entrées.

**Définition 5.** *Le fuzzing consiste à exécuter de manière répétée un programme avec des entrées qui sont syntaxiquement ou sémantiquement intentionnellement malformées.*

*Stricto sensu*, il n'y a pas de différence entre le fuzzing et du test logiciel, néanmoins le fuzzing référence dans l'usage, le scénario où le test est effectué par un attaquant avec l'objectif de trouver des vulnérabilités. Ainsi, le fuzzing implique une notion de violation de politique de sécurité.

On distingue deux catégories de fuzzers, les fuzzers guidés par la couverture du programme qu'ils produisent (*coverage-guided*) et ceux orientés par un but ou un emplacement du programme à atteindre (*directed fuzzing*). L'écrasante majorité des fuzzer est guidée par la couverture. À la marge des travaux propose des métriques de guidage tel que le *stack-depth-guided*, *intensity-guided* ou encore la métrique *allocation-guided*<sup>115</sup>.

---

114. <https://habr.com/en/company/dsec/blog/449134/>

115. <https://guidovranken.com/2017/07/08/libfuzzer-gv-new-techniques-for-dramatically-faster-fuzzing/>

La littérature concernant les techniques de fuzzing est abondante. Le présent état de l'art s'appuie fortement sur les travaux effectués par Valentin J.M Manès au KAIST [246]. La communauté travaillant sur le fuzzing est telle qu'il est difficile de suivre et de repérer les réelles avancées dans ce domaine. De plus, la terminologie utilisée n'est pas forcément unifiée à travers les travaux de recherche. Par exemple, l'action de réduction de la taille d'une entrée provoquant un crash est tantôt qualifiée de "minimisation de crash", tantôt de "réduction de cas de test". Cet état de fait complexifie l'étude des différentes ressources.

Ce document vise à présenter l'ensemble du spectre des différentes approches tout en se focalisant, pour une analyse détaillée, sur les outils prometteurs dans le cadre d'un démonstrateur. Il existe trois grands types de fuzzers :

**Fuzzer black-box** c'est l'approche la plus simple dans laquelle l'utilisateur n'a aucun accès aux mécanismes internes du programme. L'algorithme peut simplement observer les entrées/sorties avec un accès "dit" oracle. Cette technique peut généralement être mise en oeuvre facilement, mais elle reste la moins susceptible de trouver des bugs. De nombreux fuzzers entrent dans cette catégorie (BFF [348], Radamsa [172], PULSAR [132] etc.).

**Fuzzer white-box** ce terme, introduit par Godefroid [136] en 2007, désigne les algorithmes utilisant une connaissance sémantique du fonctionnement du programme pour générer les cas de test. Il désigne donc plus communément les moteurs d'exécution concoliques utilisés pour de la recherche de vulnérabilité.

**Fuzzer grey-box** dans cet intermédiaire entre le fuzzing *black-box* et *white-box*, l'algorithme a une certaine connaissance du fonctionnement interne du programme et s'en sert par exemple pour récupérer de l'information à chaque exécution afin d'améliorer les exécutions ultérieures. Ces fuzzers sont dit *feedback-driven*. C'est le cas par exemple des fuzzers utilisant la couverture du programme à chaque exécution pour générer des cas de tests avec des algorithmes évolutifs (cf. 9.7). La plupart des fuzzers modernes tels que AFL, VUzzer [290] ou Honggfuzz [335] entrent dans cette catégorie.

Les fuzzers *white-box* étant au sens propre des moteurs de DSE munis des différentes fonctions d'un fuzzer, ils seront traités dans cet état de l'art comme des combinaisons de DSE et Fuzzing (*noté CF*). Cette technique est décrite dans le chapitre 12 et plus particulièrement la section 12.3.

## 9.2 Algorithme

Le processus de fuzzing – quel que soit le type de fuzzer – se décompose en différentes fonctions effectuant chacune une tâche précise (cf. Algorithme 2). Les principales fonctions sont **Preprocess**, **Schedule**, **InputGen**, **InputEval**, **ConfUpdate** et **Continue**. L'algorithme 2 prend en entrée un programme  $P$ , un ensemble de configurations initiales  $\mathbb{C}$ , un *timeout* noté  $t_{limit}$  et un oracle de bug permettant de décider si un comportement

est ou non caractéristique d'une violation de la politique de sécurité (cf. 9.6.2). On note  $c \in \mathbb{C}$  une configuration représentant un état courant de couverture à partir d'une graine d'entrée donnée. En sortie, l'algorithme retourne un ensemble de bugs notés  $\mathbb{B}$  symptomatiques d'une violation de la politique de sécurité telle qu'elle est définie par l'oracle de bug  $\mathbb{O}_{bug}$ .

---

**Algorithm 2** Algorithme de fuzzing

---

**Require:**  $P, \mathbb{C}, t_{limit}, \mathbb{O}_{bug}$

```

1:  $\mathbb{B} \leftarrow \emptyset$ 
2:  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3: while  $t_{elapsed} < t_{limit} \wedge \text{Continue}(\mathbb{C})$  do
4:    $c \leftarrow \text{Schedule}(\mathbb{C}, t_{elapsed}, t_{limit})$ 
5:    $tcs \leftarrow \text{InputGen}(c)$ 
6:    $b, exec_i \leftarrow \text{InputEval}(P, tcs, \mathbb{O}_{bug})$ 
7:    $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, c, exec_i)$ 
8:    $\mathbb{B} \leftarrow \mathbb{B} \cup \{b\}$ 
9: end while
10: return  $\mathbb{B}$                                  $\triangleright$  l'ensemble des bugs trouvés

```

---

Les différentes fonctions assurées par un algorithme de fuzzing sont :

**Preprocess( $\mathbb{C}$ )  $\rightarrow \mathbb{C}$**  : appelée une fois par campagne, cette fonction a pour but, à partir d'un ensemble d'entrées (noté  $TCS$ ) correspondant à un jeu de fichiers ou autres, de fournir un ensemble de configurations initiales auxquelles sont associées des graines.

**Schedule( $\mathbb{C}, t_{elapsed}, t_{limit}$ )  $\rightarrow c$**  : a pour but de sélectionner une configuration dans l'ensemble des configurations en attente. La principale difficulté de cette fonction est de prioriser les configurations prometteuses en termes de résultats.

**InputGen( $c$ )  $\rightarrow tcs$**  : à partir d'une configuration donnée, fait évoluer les entrées du programme pour fournir un cas de test  $tcs$  permettant, selon la stratégie, d'améliorer la couverture ou de déclencher un bug.

**InputEval( $P, tcs, \mathbb{O}_{bug}$ )  $\rightarrow b, exec_i$**  : fonction d'exécution concrète du programme avec les entrées  $tcs$ . Cette fonction retourne potentiellement un bug  $b$  ainsi que des informations supplémentaires sur l'exécution qui, dans le cas de fuzzing *grey-box*, servent dans la boucle de rétroaction. Les informations sont généralement des informations de couverture.

**ConfUpdate( $\mathbb{C}, c, exec_i$ )  $\rightarrow \mathbb{C}$**  : sert à mettre à jour la configuration courante ou plusieurs autres configurations à partir des informations récoltées lors de l'exécution.

**Continue( $\mathbb{C}$ )  $\rightarrow \{\text{True}, \text{False}\}$**  : définit le critère d'arrêt de la campagne de fuzzing. Cela peut être un taux de couverture défini par une fonction de couverture (e.g : couverture des branches, basic-blocks) ou plus simplement un *timeout* global pour la campagne. Dans le cadre du projet, ce peut être un taux de couverture des objectifs de tests (*Test Requirements* noté  $TR$ ).

L'implémentation de ces différentes fonctions varie énormément d'une approche et d'un outil à l'autre. Les sections suivantes décrivent plus précisément le fonctionnement

de ces différentes fonctions parmi tous les outils examinés pour cet état de l'art.

## 9.3 Preprocessing

La phase de préprocessing (fonction **Preprocess**) a pour rôle de créer les configurations initiales et d'effectuer toutes les initialisations nécessaires aux étapes suivantes. Cela peut-être l'ajout de code d'instrumentation ou toute autre analyse statique ou dynamique nécessaire au fonctionnement de l'algorithme.

### 9.3.1 Instrumentation

À l'exception du fuzzing *black-box*, les autres approches ont recours à l'instrumentation pour obtenir des informations sur l'exécution, la couverture ou pour jouer le rôle d'oracle de bug. L'instrumentation est soit statique (elle requiert généralement le code source), soit dynamique et peut donc fonctionner sur du binaire.

**Instrumentation statique.** Elle est effectuée sur le programme avant qu'il soit exécuté. Elle peut s'effectuer sur le code source lors de la compilation, comme c'est le cas pour AFL et ses dérivés, mais elle peut aussi être faite au niveau binaire grâce à de la réécriture binaire (*binary rewriting*) [222, 330, 392]. Ce dernier cas est généralement plus hasardeux, car il requiert un désassemblage correct du programme. Dans les deux cas, l'avantage de l'instrumentation statique est qu'elle est généralement plus rapide, car directement intégrée au code.

**Instrumentation dynamique.** Elle s'effectue au lancement du programme via l'interception du flux de contrôle du programme. Ceci à l'avantage de pouvoir instrumenter facilement toutes les bibliothèques partagées utilisées par un programme. Évidemment le processus de lancement du programme est nécessairement ralenti, or cette composante est critique pour le fuzzing. Les outils les plus connus sont Pin [241] (utilisé par Choronzon [396]), DynamoRIO [48], QEMU [26], Valgrind [264], DynInst [101], mais aussi QBDI [181] développé à Quarkslab<sup>116</sup>.

L'instrumentation, qu'elle soit statique ou dynamique, est dépendante de l'architecture cible. En effet, même si l'instrumentation est effectuée au niveau source, le code d'instrumentation injecté utilise systématiquement des fonctionnalités spécifiques à l'architecture cible. Le tableau 9.1 résume les différentes capacités des méthodes d'instrumentation rencontrées dans l'état de l'art.

---

116. <https://qbdì.quarkslab.com/>

117. utilisé pour du DSE

	Source	Niveau		Instr.	Architectures				
		Binaire	Statique		Dynamique	x86	x86-64	ARMv7	ARMv8
AFL [383] ( <i>like</i> )	✓		✓			✓	✓	✗	✗
DynamoRIO [48]		✓		✓		✓	✓	✓	✓
DynInst [101]		✓		✓		✓	✓	✗	~
PEBIL [222]		✓	✓			✓	✓	✗	✗
PSI [392]		✓	✓			✓	✗	✗	✗
QBDI [181]		✓	✓	✓		✓	✓	~	✗
QEMU [26]		✓		✓		✓	✓	✓	✓
Valgrind [264]		✓		✓		✓	✓	✓	✓
Vulcan [330]		✓	✓			✓	✓	✗	✗
iDNA <sup>117</sup> [31]		✓		✓		✓	✓	✗	✗

TABLE 9.1 – Comparatifs d’outils d’instrumentation

### 9.3.2 Construction de l’ensemble de configurations initiales

Avant exécution, un ensemble des fichiers d’entrée peut être fourni au fuzzer. Généralement, plus il est représentatif des différentes entrées possibles du programme, plus la campagne de fuzzing sera fructueuse. Une des tâches de la fonction **Preprocess** est de traiter cette suite de test pour créer un ensemble de configurations qui seront utilisées ensuite. Il convient donc de sélectionner et d’élagger cette suite de test pour qu’elle soit suffisamment synthétique. Ces deux étapes sont aussi effectuées par la fonction **ConfUpdate** et sont décrites dans la section mise à jour des configurations 9.7.

## 9.4 Ordonnancement

L’ordonnancement (fonction **Schedule**) consiste à choisir une configuration  $c$  dans l’ensemble des configurations  $\mathbb{C}$ . Le but est de prioriser les configurations les plus susceptibles de fournir le résultat attendu. Or, le résultat attendu peut varier. Cela peut-être de maximiser la couverture, de trouver le plus de bugs uniques ou de couvrir les objectifs de test insérés dans le programme. On appelle cette fonction la fonction d’objectif *objective function* (notée  $\mathbb{F}$ ). D’un point de vue mathématique, le seul but du fuzzing est de maximiser cette fonction et, pour y parvenir, le fuzzer doit sélectionner adroitemment la configuration la plus adaptée pour la prochaine exécution. Ce problème est connu sous le nom de *Fuzzing Configuration Scheduling (FCS) problem* et sera référencé par RQ\_F1. Certains outils comme **zzuf** [174] n’utilisent qu’une configuration et n’ont pas ce problème,

mais d'autres outils tels que **AFLfast** [39] concentrent une grande partie de l'intelligence du fuzzer dans cette fonction en catégorisant les chemins par la fréquence avec laquelle ils sont exercés. L'approche pour résoudre ce problème dépend cependant beaucoup du type du fuzzer.

**Approche black-box** ; dans cette approche, les seuls métriques exploitables pour prioriser une configuration plutôt qu'une autre sont le nombre de bugs trouvés et le temps d'exécution d'une configuration. Tandis que l'approche neutre effectue un échantillonnage uniforme des configurations *uniform sampling*, une approche plus intuitive est de favoriser les configurations qui ont le meilleur ratio. Celle-ci fut explorée par Foote et al [180] qui ont proposé un ordonnancement des mutations en boîte noire basé sur une séquence d'épreuves de Bernoulli. Woo et al ont proposé une solution basée sur le problème du collectionneur de vignettes à poids inconnus *Weighted Coupon Collector's Problem with Unknown Weights* abrégé WCCP/UW [365]. Dans cette même publication ils ont aussi proposé une autre approche basée sur le problème du bandit manchot *Multi-armed bandit* abrégé MAB. Cette technique leur permet d'optimiser le temps alloué à chaque configuration en normalisant la probabilité de succès d'une configuration en se basant sur le temps qui a déjà été passé avec elle. Ce procédé probabiliste leur permet de limiter le temps à passer dans les configurations "pourries" "*decayed*".

**Approche grey-box** ; ce type de fuzzer peut pleinement tirer parti des connaissances internes acquises au fur et à mesure des exécutions pour prioriser des configurations. **AFL** fait figure d'exemple en utilisant un algorithme génétique ([Evolutionary Algorithm \(EA\)](#)) qui utilise une valeur de *fitness* sur chaque configuration après avoir préalablement appliqué les mutations permettant d'obtenir l'entrée mutée (descendante). Plus une entrée est compacte et rapide à exécuter, plus elle est favorisée par rapport aux autres. **VUzzer** utilise une fréquence d'exécution des basic-block pondérée. **AFLfast** ajoute deux critères à cette fonction qui sont : 1) pour un chemin donné, favoriser la configuration qui a été choisie le moins souvent, 2) favoriser les configurations dont le chemin a été le moins exercé. Ces deux critères visent à encourager la couverture de chemins peu empruntés. **AFLfast** introduit aussi la notion d'énergie *power schedule*, une configuration démarre avec une faible énergie qui augmente progressivement en fonction du nombre d'exécutions et du nombre d'entrées exerçant ce chemin.

Pour cette fonctionnalité, une approche intéressante est celle de **QTEP** [355] et **AFLGo** [40] qui met en oeuvre une phase d'analyse statique pour déterminer les zones sensibles et susceptibles d'être vulnérables, puis pour choisir les configurations les plus aptes à couvrir ces zones.

## 9.5 Génération des entrées

La fonction de génération des entrées (**InputGen**) est sans doute la fonction la plus critique dans un fuzzer. Il existe traditionnellement deux types d'algorithmes, ceux basés

sur un modèle de format (dit génératifs) et ceux qui se basent sur de la mutation.

### 9.5.1 Génération par modèle

La génération par modèle est particulièrement efficace, car elle permet de fuzzer uniquement les portions pertinentes d'un bloc binaire. L'inconvénient est que cela nécessite une connaissance du format, ce qui fait perdre l'aspect portable du fuzzer qui se spécialise sur un modèle particulier. La plupart des outils fournissent une API ou une grammaire permettant d'exprimer un format à partir duquel le fuzzer sera en mesure de générer des mutants. Parmi ces outils, on peut noter [Peach](#) [115] et [Radamsa](#) [172]. Récemment, un intérêt particulier a été donné pour modéliser des formats utilisés dans les navigateurs comme DOM avec [cross\\_fuzz](#) [384] ou encore Javascript avec [jsfunfuzz](#) [308]. Des approches plus poussées proposent des modèles non seulement grammaticalement (syntaxiquement) corrects, mais aussi sémantiquement corrects. C'est le cas notamment pour [LangFuzz](#) [176] et [BlendFuzz](#) [374], [zest](#) [271] fonctionnant respectivement sur Javascript/[PHP Hypertext Preprocessor \(PHP\)](#) et sur [Extensible Markup Language \(XML\)](#). Le Javascript suscite actuellement un intérêt particulier pour son utilisation dans les navigateurs web. Ainsi, [Google Project Zero](#) a récemment publié [fuzzilli](#) [151] dédié aux moteurs de Javascript.

Appliquée aux langages de programmation, [universalmutator](#) [150] propose un algorithme de mutation de code source basé sur des expressions régulières pouvant être appliquées, dans la logique, à n'importe quel langage. La principale difficulté dans ce contexte est de générer un programme qui compile.

Un des problèmes rencontrés avec la génération par modèle est que l'algorithme de mutation prend rarement en compte l'efficacité de couverture produite par le mutant pour la mutation des futurs descendants. L'outil [Nautilus](#) [12] publié récemment propose un couplage fort entre l'algorithme de génération des entrées par grammaire et le *feedback* obtenu de l'exécution de l'entrée.

#### Modèle de protocole

L'usage de grammaires pour modéliser les protocoles réseaux est particulièrement pertinent pour du fuzzing *black-box* à travers le réseau comme c'est le cas pour [T-Fuzz](#) [192], [SNOOZE](#) [21], [PROTOS](#) [198] ou encore [Kif](#) [3] pour le protocole SIP. Cependant, le fuzzing de protocole réseau nécessite plus qu'un simple modèle du protocole (cf. section 9.8).

#### Modèle d'appel système

Beaucoup de fuzzers sont dédiés au fuzzing des Application Binary Interface (ABI)s noyau qui, si elles aboutissent, peuvent potentiellement mener à des escalades de priviléges. Les fuzzers connus utilisant ce type de modèle sont [Trinity](#) [194], [KernelFuzzer](#) [218],

Triforce [153] ou encore `syzkaller` [352]. Dans ce contexte les modèles représentent l'ABI entre le mode utilisateur et le mode kernel de l'OS. Ce type de modèle et les fuzzers associés seront écartés pour la suite de cet état de l'art. `syzkaller` s'est récemment illustré [212] pour sa capacité à fuzzer les drivers USB du noyau Linux sur lesquels pas moins de 80 bugs ont été trouvés<sup>118</sup>.

## Inférence automatisée du modèle

L'inférence automatisée du modèle a suscité un intérêt particulier dans le monde de la recherche (RQ\_F2) et en particulier pour les fuzzers de protocole réseau [55, 85]. Cette inférence peut être faite au niveau **Preprocess** via l'analyse statique ou dynamique du programme [94, 239] ou bien dans **ConfUpdate** à partir des entrées/sorties générées.

**En prétraitement.** Parmi les fuzzers effectuant l'inférence comme une passe de pré-traitement on peut noter `Skyfire` [353] qui analyse des données (*dataflow*) à partir d'un échantillon d'entrées pour apprendre une grammaire probabiliste sensible au contexte *Probabilistic Context-Sensitive Grammar (PCSG)*. `Test-Miner` [344] utilise quant à lui les informations du code pour effectuer son inférence.

**Par machine learning.** Le machine learning s'est récemment introduit dans le domaine du fuzzing appliqué majoritairement à l'inférence de modèle. Ainsi, `Neural` [81] et `NEUZZ` [316] proposent d'utiliser un réseau de neurones pour apprendre la grammaire à partir d'un jeu d'entrées. Des travaux prometteurs sont ceux effectués par Patrice Godefroid, publiés en 2017 avec `Learn&Fuzz` [141], qui explorent différentes méthodes à base de `Recurrent Neural Network (RNN)` et de `Long Short-Term Memory (LSTM)`. Enfin récemment `V-Fuzz` [236] mélange un modèle de prédiction de vulnérabilité basé sur un réseau de neurones entraîné avec Pytorch<sup>119</sup>, accompagné de la notion d'*Attributed-CFG* représentant un CFG où chaque basic block est un vecteur d'attributs.

**Pendant la mise à jour des configurations.** Une autre approche est d'apprendre le modèle au fur et à mesure des exécutions. Cette méthode évite de devoir disposer d'un jeu d'entrées significatif. De cette manière, `GLADE` [25] synthétise le modèle à partir de couples d'entrées-sorties. Encore une fois, pour les protocoles réseau, la mise à jour à partir des paquets transmis sur le réseau permet d'inférer dans une certaine mesure les formats des échanges. `PULSAR` [132] effectue cette inférence et effectue la corrélation des messages avec la machine à états du protocole construite progressivement (voir section 9.8). Enfin `ProFuzzer` [378] propose une approche d'"*input probing*" dont le but est de déterminer le type des différents champs d'un bloc binaire pour adapter dynamiquement les techniques de mutation.

---

118. [https://github.com/google/syzkaller/blob/usb-fuzzer/docs/linux/found\\_bugs\\_usb.md](https://github.com/google/syzkaller/blob/usb-fuzzer/docs/linux/found_bugs_usb.md)

119. <https://pytorch.org/>

### 9.5.2 Génération par mutation

L'élément clé de la génération *mutation-based* est l'utilisation d'une graine *seed* à partir de laquelle est dérivée une entrée. Une graine est généralement un fichier ou un paquet réseau bien structuré à partir duquel est dérivée l'entrée malformée par mutation. Il existe un grand nombre de techniques de mutation dont voici les plus couramment utilisées :

**Inversion de bits** : *bit-flipping*, c'est une technique très simple à mettre en oeuvre, car il suffit d'inverser des bits arbitraires de la graine pour générer une nouvelle entrée. AFL, Honggfuzz, Radamsa et EFS [100] utilisent intensément cette technique. Certains utilisent aussi un ratio de mutation pour muter une quantité de bits proportionnelle à la taille de la graine. Ce ratio a un impact sur les performances [63], néanmoins la valeur optimale change d'un type d'entrée à un autre. L'évolution de ce ratio reste à la discréption de l'outil. Certains outils comme BFF [348] et FOE [349] utilisent par défaut une échelle exponentielle qui peut ensuite être adaptée pour les configurations efficaces (cf. 9.4), SymFuzz [63] à l'inverse, infère le ratio à partir d'une exécution symbolique du programme pour déterminer les ratios adaptés.

**Dictionnaire** : utilisation d'un ensemble de valeurs prédéterminées tirées dans un dictionnaire. Pour des entiers, les valeurs sont souvent 0, -1 et 1 (valeurs limites), tandis que pour des chaînes de caractères, des caractères unicode ou des formateurs "%s", "%x".

**Mutation arithmétique** : mutation employée par AFL et appliquée sur les octets considérés comme un entier par le fuzzer. À cet effet, AFL sélectionne des couples de 4 octets traités comme un entier auquel est incrémenté ou décrémenté une valeur choisie aléatoirement.

**Mutation de blocs** : *block-based mutation*, elle vise à effectuer des opérations sur des séquences d'octets de la graine [172, 335, 338, 383]. Les principales actions effectuées sur les blocs sont :

1. insertion d'un bloc aléatoire à une position aléatoire de la graine
2. suppression d'un bloc aléatoire
3. permutation aléatoire d'une séquence de blocs
4. ajout d'un bloc aléatoire à la fin
5. insertion ou remplacement d'un bloc par celui d'une autre graine

La sélection et l'ordonnancement des stratégies de mutation restent un problème de recherche ouvert. Il sera référencé par la question de recherche RQ\_F3 dans la suite du document. De multiples études empiriques ont cependant été effectuées pour déterminer les mutations efficaces <sup>120</sup>.

---

120. <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>

## 9.6 Exécution du programme

L'exécution du programme (**InputEval**) sur la configuration sélectionnée est la plus gourmande en temps, car elle requiert de lancer une nouvelle instance du programme avec l'entrée sélectionnée. Les deux problèmes qui se posent sont de rendre ce processus aussi rapide que possible et de détecter toute violation de la politique de sécurité définie par l'oracle de bug  $\mathbb{O}_{\text{bug}}$ .

### 9.6.1 Optimisations

#### Serveur Fork

La principale optimisation introduite pour améliorer le temps de lancement d'un programme fut celle du *forkserver* par AFL. Le principe est d'économiser le temps de chargement d'un programme et de toutes ses bibliothèques partagées en lançant chaque nouvelle instance avec l'appel système Linux `fork()`. Xu et al. [371] ont par ailleurs proposé de pousser encore plus loin l'optimisation en implémentant un nouvel appel système similaire à `fork` mais éludant certaines vérifications.

Sous Windows, cette technique n'existe pas, car l'appel système n'existe pas. Plus précisément, il existe au niveau noyau, mais aucune API *userland* ne permet d'y accéder. De cette manière WinAFL [390] est forcée de générer un nouveau processus de la manière habituelle (pour compenser il utilise intensément le mode persistent décrit ci-dessous). Sienna Locomotive subit aussi cette limitation étant lui aussi spécialisé pour les programmes Windows.

#### In-Memory Fuzzing

Cette optimisation utilisée par AFL et ses dérivés permet de fuzzer une configuration donnée sans avoir à recréer un nouveau processus. Ceci fait sens pour les applications graphiques ou pour les programmes utilisant le réseau qui nécessitent souvent une longue phase d'initialisation. Une solution mise en oeuvre par GRR [35] est d'effectuer un *snapshot* des registres et de la mémoire avant chaque itération. Ceci est potentiellement plus rapide que le `fork` mais limite beaucoup l'intérêt du fuzzing en mémoire. L'autre solution utilisée par `libfuzzer` et AFL est tout simplement de conserver la mémoire telle qu'elle est. Cette option s'appelle mode “persistent” dans AFL. Le principal inconvénient et que cela casse la correction des résultats (l'entrée utilisée ne sera potentiellement pas capable de recouvrir le chemin activé). Cette solution pose donc un problème de reproductibilité pourtant critique en fuzzing.

## Cassage de comparaisons

Lors d'une campagne de fuzzing, la découverte de chemins (et donc de bugs) peut se trouver freinée par des problèmes difficilement solubles par un fuzzer simpliste. En effet, un des éléments bloquants est la rencontre d'une comparaison avec un [magic-number](#). Ce regroupement d'octets arbitraires peut mettre en défaut les fuzzers disposant de mutations trop simplistes (question de recherche RQ\_F4). En effet, prenons l'exemple d'une valeur arbitraire sur 4 octets : **0xdeadbeef**. Une mutation aléatoire des bits de l'entrée ne permettra pas de satisfaire simplement la comparaison, conduisant à une recherche exhaustive d'un entier sur un espace de 32 bits. Une solution, démocratisée par [235], permet de *casser* efficacement ces comparaisons, en obtenant un retour plus précis lors de la comparaison. Au lieu d'effectuer une seule comparaison, celle-ci va être divisée en de multiples comparaisons d'entiers, le fuzzer est alors averti à chaque fois qu'une de ces comparaisons est validée, permettant ainsi d'obtenir des résultats pertinents bien plus rapidement. Par exemple, en divisant une comparaison d'entier sur un espace de 32 bits (4 294 967 296 possibilités), l'on effectue 4 comparaisons de 8 bits ( $4 * 256 = 1024$  possibilités). La recherche exhaustive de la solution est alors nettement plus efficace, permettant au fuzzer de découvrir des chemins jusqu'ici difficilement explorables. [LAF-intel](#) [219] effectue automatiquement cette scission lors de la compilation du programme ce qui permet d'utiliser un fuzzer n'ayant pas nécessairement cette optimisation.

### 9.6.2 Oracles de bugs

Les oracles de bugs servent à détecter différents types de bugs de différentes manières et avec une précision qui a généralement un impact direct sur la rapidité d'exécution du programme. Nonobstant cet usage, un oracle peut détecter n'importe quel événement révélateur d'une violation de la politique de sécurité à vérifier.

#### Sureté mémoire

Une méthode standard de détection d'un bug consiste à détecter un signal de crash. Cette méthode permet de détecter une corruption mémoire via le déréférencement d'un pointeur sur une page mémoire invalide. Pour autant, cette méthode ne détecte une corruption que si celle-ci déclenche ensuite un déréférencement invalide. Des corruptions mémoires peuvent avoir lieu sans que cela déclenche un crash. Pour parer ce problème, différentes transformations de code ont été proposées afin de détecter ce genre d'événement de manière efficace dans ce que l'on appelle un *sanitizer* (sonde d'assainissement). Les corruptions mémoires se catégorisent en deux types :

**Corruption spatiale** : elle se produit lorsqu'un pointeur est accédé en dehors de son intervalle d'adresses. Les débordements de tampon ou les off-by-one en sont un bon exemple.

**Corruption temporelle** : elle se produit lorsqu'un pointeur est accédé alors qu'il n'est plus valide, comme pour les use-after-free.

*AddressSanitizer* [312] (*ASan*) ajoute du code d'instrumentation à la compilation pour détecter les deux types de corruptions susmentionnées. Il utilise pour cela une *shadow memory* pour détecter à chaque déréférencement si le pointeur est valide. Il ne permet pas de détecter tous les cas d'accès impropre, mais il en couvre une bonne partie – y compris ceux ne déclenchant pas de bug – et ce au prix d'un ralentissement acceptable. Aussi développé par Google, *LeakSanitizer* [224] enrichit *ASan* en permettant aussi la détection de fuites mémoires. D'autres outils visent à améliorer les performances de détection. Par exemple, l'outil *MEDS* [162] améliore *ASan* avec une propriété de tas (*heap*) infini permettant d'améliorer dans certains cas de 68,3% le taux de détection de corruptions au prix d'un ralentissement de 108%. Le couple *SoftBound* [260] et *CETS* [261], développés par les mêmes chercheurs apporte respectivement une vérification spatiale et temporelle des accès mémoires. Il permet théoriquement de détecter 100% des corruptions possibles par le suivi exact des bornes et des valeurs possibles pour chaque pointeur du programme au prix d'un ralentissement de 116%.

## Sécurité des flots de contrôle et de données

Ce type de *sanitizer* cible la sécurité des deux différents flots d'un programme à savoir le **CFG** et le **Data Flow Graph (DFG)**. Pour les **CFG**, le **Control Flow Integrity (CFI)** [2] vérifie l'intégrité des pointeurs impliqués dans le flot de contrôle du programme. C'est une vérification plus stricte de l'assainissement de mémoire, car, en plus de vérifier la validité des pointeurs, elle s'assure que les valeurs possibles sont celles attendues par le **CFG**. Cette protection s'applique généralement aux sauts et aux appels dynamiques et permet donc de détecter les détournements de flot de contrôle. Pour les **DFG**, *DataFlow-Sanitizer* [96] permet au programmeur de “tagger” des données. Les tags seront ensuite propagés dynamiquement en fonction des opérations effectuées sur les données pour détecter des comportements non voulus par le programmeur. À l'inverse des précédents *sanitizer*, celui-ci ne détecte pas une classe spécifique de bugs, mais vérifie la conformité du **DFG** à un comportement spécifié par le programmeur. Ceci revient à fournir l'oracle sur lequel sera vérifiée dynamiquement la conformité du **DFG**. Cette technique peut être utilisée pour faire de l'analyse de flots d'informations.

## Coercition de type

Ce type de défaut, également appelé *bad casting*, est un type d'erreur pouvant se produire lorsque le compilateur n'a pas été en mesure de le vérifier a priori. Ainsi, le code d'instrumentation ajouté peut vérifier de mauvais *cast C++* d'un type d'objet à un autre. *TypeSan* [158] et *HexType* [159] effectuent ce type de détection ainsi que *CaVer* [226] qui s'est fait remarquer pour son passage à l'échelle sur des navigateurs.

## Comportements indéfinis

Certains langages, en particulier le langage C, présentent un certain nombre de comportements indéfinis *undefined behavior* dans la norme. Bien que n'impliquant pas directement une vulnérabilité, le comportement est laissé à la discréption du compilateur et cela mène potentiellement à des comportements inattendus et donc dangereux. À cet effet, *MemorySanitizer* [332] (*MSan*) détecte exclusivement l'utilisation de mémoire non initialisée (avec le même mécanisme de *shadow-memory*). À l'inverse, *UBSan* [103] détecte différents comportements indéfinis comme les pointeurs désalignés, les divisions par zéro, le déréférencement de pointeur nul et les débordements d'entiers signés.

## Sanitizer de thread

Il existe très peu de *sanitizers* permettant de détecter des corruptions liées à la concurrence (*data-race*). On considère qu'il s'en produit une lorsque deux threads accèdent à une même ressource en parallèle et que l'un d'eux y accède en écriture. Les bugs générés sont très difficiles à reproduire à cause du non-déterminisme que cela implique. À ce jour, le seul outil effectuant ce type d'assainissement est *ThreadSanitizer* [313].

## Test différentiel

Le test différentiel est orthogonal aux autres méthodes plus classiques de détection de bugs. En effet cette méthode ne détecte pas des bugs, mais des déviations de comportement et de résultats par rapport à d'autres programmes équivalents [75, 196]. Ceci permet, bien au-delà de la recherche de vulnérabilités, de faire des tests de non-régression ou des tests fonctionnels sur le programme. C'est la seule méthode permettant de détecter des bugs logiques dans l'implémentation d'algorithmes de référence. Cette approche est de plus en plus utilisée [279] et en particulier pour les implémentations d'algorithmes cryptographiques, comme c'est le cas pour CDF [14]. Elle permet aussi de tester les attaques par canaux auxiliaires (*side-channel*) en temps ou en espace [265].

### 9.6.3 Triage des bugs

Le triage de bug est un problème essentiel des fuzzers, car deux crashes à deux endroits différents peuvent dans les faits être issus de la même cause. Il faut donc limiter les bugs redondants et également les ordonner si possible par criticité. Cette fonction retournant les sorties de la campagne de fuzzing, celles-ci doivent être aussi exploitables que possible pour l'analyste.

## Déduplication

Le rôle de la déduplication est de conserver un unique cas de test pour chaque bug trouvé tout en faisant en sorte que ce bug soit le plus exploitable ensuite. La difficulté reste

d'identifier l'origine *root-cause* d'un crash. Cette problématique de recherche notée RQ\_F5 est prépondérante pour les fuzzers produisant un grand nombre de cas de tests. La méthode la plus utilisée est le hachage de la pile d'appels (*stack backtrace hashing*) permettant d'obtenir un chemin relativement significatif et discriminant. La profondeur considérée est différente en fonction des outils, de un pour *Crash Wrangler* [184], à trois [252, 365] et à cinq pour BFF [348] et *exploitable* [124]. Certains fuzzers n'ont même aucune limite comme *Mutagen* [201]. L'autre méthode utilisée par beaucoup de fuzzers *grey-box* est l'utilisation de la couverture du chemin pour discriminer des crashes. AFL utilise cette technique et considère un crash comme unique si le chemin couvre une nouvelle arête ou s'il ne couvre pas une arête présente dans les précédents crashes. Une dernière approche, similaire à celle employée par Triton [304] pour du DSE, est celle employée par RETracer [93] qui trie les bugs en effectuant une analyse *data-flow* inversé – que l'on peut raisonnablement appeler slicing – pour suivre le pointeur fautif jusqu'à la valeur ayant effectuée l'assignation.

## Minimisation de cas de test

La minimisation de cas de test vise à identifier la portion de l'entrée responsable du crash afin de, si possible, réduire la taille totale de l'entrée. Le but est d'avoir le cas de test minimal pour améliorer les mutations subséquentes effectuées sur l'entrée. La plupart des fuzzers intègrent leurs propres heuristiques pour réduire le cas de test. Par exemple, AFL essaye de remplacer des octets par zéro tout en réduisant la taille du fichier. À l'inverse l'algorithme [179] de BFF tente de différencier les octets de la graine initiale qui ont changé. Enfin des “minimiseurs” de cas de tests sont utilisés en compilation et pas uniquement en fuzzing. Parmi ceux-ci, on peut citer *lithium* [298], développé par Mozilla, qui utilise un test de *interestingness* utilisé en *delta-debugging*<sup>121</sup>. En sus, cette implémentation se base sur l'algorithme de *ddmin* [389].

## Priorisation & Exploitabilité

Une fois triés et minimisés, les cas de tests retenus doivent de préférence être triés pour faciliter l'analyse des résultats. Ce problème, connu sous le nom de *taming problem* [74], est celui de regrouper les bugs par unicité et par sévérité (noté RQ\_F6). Le principe étant d'ordonner par exploitabilité et criticité. Le premier outil à prioriser des crashes fut *!exploitable* [89] écrit par Microsoft dans *WindDBG* et fut rapidement suivi par *Crash Wrangler* et *exploitable* dans GDB. Toutes ces approches utilisent différentes heuristiques et, pour certaines, l'analyse de teinte. Cet axe de recherche, l'un des moins explorés, est l'un des plus importants dans le domaine du fuzzing.

---

121. <https://www.st.cs.uni-saarland.de/dd/>

## 9.7 Mise à jour des configurations

### 9.7.1 Sélection des graines

Ce problème, connu sous le nom de *seed selection problem* (noté RQ\_F7), est celui de trouver l'ensemble minimal de graines représentatives de l'espace des entrées possibles du programme. Cet espace étant usuellement infini, il convient de réduire ces graines à un nombre de configurations restreint. L'approche la plus utilisée consiste à exercer chaque entrée sur le programme et ne conserver que celles qui maximisent la couverture [115, 291]. La logique sous-jacente est que le nombre de bugs augmente avec la couverture. Honggfuzz [335] utilise une couverture des instructions, de branches exécutées et de basic-block uniques pour sélectionner ces graines. Cela favorise les exécutions plus profondes. Cette sélection est effectuée sur toutes les graines dans le prétraitement (cf. 9.3) et lors de chaque exécution, après avoir exercé l'entrée sur le programme.

### 9.7.2 Mise à jour des graines

Une fois la graine exercée et mutée, elle est remise en queue de l'ensemble des configurations en attente. Certains fuzzer essayent de les couper pour être plus compacts. AFL utilise la couverture pour couper des portions de graines qui après mutation n'ont rien changé au chemin emprunté. À noter qu'une plus petite graine ne produit pas forcément une meilleure couverture, il faut donc élaguer avec précaution [291].

#### Mise à jour évolutionnaire

L'Algorithme *Evolutionary Seed Pool Update*, démocratisé par AFL/libfuzzer, est un algorithme génétique (issu de la biologie) utilisé pour faire muter les entrées, mais surtout pour générer de nouvelles graines qui seront ajoutées aux configurations en attente C. Les fuzzers utilisent une fonction de *fitness* pour déterminer si le mutant doit ou non être ajouté aux configurations. Cette fonction est ce sur quoi la recherche s'oriente actuellement. AFL utilise la couverture de branches et le nombre de fois que chaque branche a été prise. Le fuzzer Angora [71] améliore cette fonction en ajoutant le contexte d'appel de chaque branche prise. VUzzer [290] quant à lui ajoute une graine aux configurations si elle couvre un nouveau basic-block non dédié à la gestion d'erreur (appelé EH *error-handling*). Cette technique se base sur le fait que les fuzzers perdent trop de temps à couvrir du code faisant de la gestion d'erreur. C'est le même facteur limitant constaté par les moteurs de DSE (cf. 6.9).

#### Mise à jour guidée

Certains fuzzers utilisent une *fitness-function* guidée par un but. Dans ce cas ils ne sont plus dirigés par la couverture, mais par un autre critère. C'est le cas de AFLGo [40] et

HawkEye [69] qui utilisent une analyse statique préalable pour être en mesure de déterminer la distance d'un chemin exercé à une cible. L'analyse se base sur la distance sur le *call-graph* entre basic-block etc.

## Mise à jour par machine learning

En marge des algorithmes évolutionnaires, différents travaux proposent des approches par machine learning pour “apprendre” comment muter la graine. Des chercheurs de Microsoft [287] proposent un réseau de neurones apprenant les motifs (*patterns*) de l’entrée (les octets) favorisant l’exploration du programme. Pour cela un modèle à base de LSTM et un modèle *sequence-to-sequence* ont été proposé.

## Minimisation des configurations

Ajouter des configurations pose nécessairement le problème d’en avoir trop. L’intérêt est donc de maintenir un ensemble minimal appelé *minset*. L’approche suivie par cyberdyne [142] est d’éliminer complètement les configurations qui ne rentrent pas dans ce *minset*. AFL utilise une méthode appelée “*culling procedure*” plus élégante qui marque comme favorables les configurations qui entrent dans ce *minset*. Celles-ci auront plus de chance d’être sélectionnées dans la fonction **Schedule**. Cette technique fournit un bon équilibre entre la taille de la queue des configurations et la diversité des cas de test.

### 9.7.3 Boucle de rétrocontrôle

En plus des configurations mises à jour et de nouvelles configurations potentiellement ajoutées, le fuzzing *grey-box* bénéficie habituellement d’informations de couverture, etc., pour mettre à jour son état global. Ces informations sont appelées *execution feedback*. AFL utilise un vecteur de bits mis à jour via une fonction de hachage prenant la couverture de branches du chemin exercé lors de l’exécution. Des collisions se produisent forcément de par la taille du bitvecteur. CollAFL [128] propose une fonction de hachage alternative qui est *path-sensitive*. À l’inverse, libfuzzer, et syzkaller utilisent une couverture de noeud (basic-block) qui, même si elle est moins précise, implique moins de collisions. Honggfuzz, lui, permet de choisir en paramètre un *feedback* basé soit sur les branches, soit sur les noeuds. Cette question de recherche de première importance sera notée RQ\_F8.

## 9.8 Fuzzing de protocoles réseau

Le fuzzing de protocoles réseaux s’effectue majoritairement en *black-box*, ce qui rend une couverture efficace du code assez difficile. Le premier problème est donc celui de l’oracle généralement très naïf pour la détection d’anomalies. Ensuite, pour donner des

résultats, les fuzzers ont généralement besoin d'un modèle du ou des protocoles impliqués pour la génération des entrées. Enfin, la plupart des protocoles sont *stateful* ce qui veut dire que leur état interne est dépendant des messages échangés précédemment. Ceci est une difficulté majeure à surmonter pour le fuzzer qui doit d'une certaine manière être sensible à l'état et aux propriétés de la machine à état (noté SM). Cela est nécessaire pour être en mesure de fuzzer efficacement les différents états de la machine sans quoi – en pur aveugle – il est très peu probable que la campagne fournit de bons résultats. Ce problème majeur vient s'ajouter aux différents problèmes mentionnés dans les sections précédentes. Enfin ce problème peut aussi être approché en *grey-box* lorsqu'on contrôle le programme et la machine/serveur sur lequel tourne le programme (cf. 9.8.4).

### 9.8.1 Problématique de l'oracle

En *black-box* à travers le réseau, aucun *sanitizer* de la section 9.6.2 ne peut-être utilisé. Les capacités de déduction de l'oracle sont très limitées. Dans un contexte d'oracle pur, en cas de problème, le fuzzer provoquera généralement une réinitialisation de la connexion, ou même n'aura aucune réponse, ce qui laisse peu de place à la déduction.

### 9.8.2 Problématique des modèles

Le fuzzing par le réseau est par essence plus lent en raison des communications nécessaires et de l'aspect très structuré des messages échangés. Cela rend essentielle l'utilisation d'une méthode à base de modèle des protocoles impliqués. En outre, cela dépend énormément des couches impliquées : au niveau de la couche 7, l'aspect structuré de l'information dépendra également du protocole mis en jeu (par exemple TFTF sera très simple, alors que TLS sera complexe). Par ailleurs, la majorité des fuzzers réseau ciblent la couche 7 comme Honggfuzz, PULSAR ou boofuzz. Peu de fuzzers attaquent les couches inférieures, on peut citer Peach [115], boofuzz [277] ou scapy\_tls [253]. À noter que des outils tels que scapy<sup>122</sup> permettent de forger programmatiquement des trames réseau arbitraires, ce qui facilite grandement le fuzzing. Quel que soit le modèle, il reste ensuite à le faire muter efficacement pour pouvoir couvrir au mieux le programme [365].

### 9.8.3 Problématique de la machine à états

En fonction de la couche protocolaire visée, les couches Ethernet, IP, ICMP, (voire UDP) sont essentiellement sans état *stateless*. En revanche les couches ISO  $\geq 4$  sont essentiellement composées de machines à états comme TCP, TLS et les protocoles de niveau 7. La prise en compte des différents états est indispensable pour couvrir au mieux les différents états.

---

122. <https://scapy.net/>

**Machine de Mealy** La meilleure abstraction d'une machine à états dans un contexte de réseau est celle des machines de *Mealy*. En théorie des automates, une machine de Mealy est un automate à états finis classique (i.e transducteur fini) effectuant une transition à chaque réception d'un message. Lors d'une transition, un message peut être émis, ce qui modélise relativement bien un protocole réseau. L'annexe A fournit des explications plus détaillées concernant les machines de Mealy.

Un transducteur fini se définit par  $\mathcal{T} = (\mathcal{Q}, q_i, \Sigma_a, \Sigma_b, \delta, \lambda)$  où  $\mathcal{Q}$  représente l'ensemble fini d'états, avec  $q_i \subseteq \mathcal{Q}$  l'état initial.  $\Sigma_a$  et  $\Sigma_b$  sont respectivement l'alphabet d'entrée et l'alphabet de sortie. La fonction  $\delta : \mathcal{Q} \times \Sigma_a \rightarrow \mathcal{Q}$  définit la fonction de transition d'un état à un autre. Enfin  $\lambda : \mathcal{Q} \times \Sigma_a \rightarrow \Sigma_b$  est la fonction de sortie qui pour un état et un message d'entrée sur l'alphabet  $\Sigma_a$  renvoie un message sur l'alphabet  $\Sigma_b$ .

Dans le contexte d'une SM à base de machine de Mealy, on appelle trace une succession d'états sur cette SM initiée généralement par une connexion et terminé par sa fermeture. Formellement, on définit une trace  $\pi_q \triangleq q_i \cdots q_n$  comme la séquence d'états produits par interprétation du message d'entrée  $S_a \triangleq a_i \cdots a_n$  avec  $a_i, a_n \subseteq \Sigma_a$  par la fonction  $\delta$  et produisant  $S_b \triangleq b_i \cdots b_n$  la séquence de message de sortie par la fonction  $\lambda$ .

Ce genre de modélisation permet d'utiliser les algorithmes connus et efficaces sur ce type d'automate. Appliquée au fuzzing, le but est de trouver le message d'entrée  $S_a$  exerçant la trace  $\pi_q$  dont la sortie  $S_b$  est symptomatique d'un bug.

**Fuzzing de machine à états** Fuzzer une machine à état quelle qu'elle soit implique de fuzzer simultanément le contenu des messages et le séquencement des messages eux-mêmes pour essayer d'entraîner la SM dans un état impropre. Cela se fait en essayant de générer une trace incorrecte en ce qui concerne la spécification du protocole. Encore faut-il que la machine à états soit spécifiée. En effet, ce n'est pas le cas pour TLS. La spécification décrit les séquences d'échanges et les messages tout en laissant une grande liberté sur la SM subséquente. Sur TLS de nombreuses vulnérabilités sur le séquencement comme FREAK ou SMACK [30] trouvés par fuzzing de la SM ont entériné la nécessité de tester les SM de protocoles.

Le fuzzing de SM nécessite souvent non seulement une implémentation de la SM, mais aussi des mécanismes permettant de réordonner le séquencement des messages. Pour le protocole TLS c'est l'approche utilisée par tous les fuzzers à savoir **TLS-Attacker** [325], **FlexTLS** [30], **tlsfuzzer** [202] ou encore **scapy\_tls** [253].

**Inférence de la machine à état** Au-delà de la problématique de fuzzing, l'inférence de SM associée à des protocoles est un sujet de recherche encore ouvert (noté ici RQ1). Sur un protocole inconnu, l'inférence d'une représentation primitive de la machine à état est nécessaire pour obtenir des résultats. Les méthodes pour y parvenir varient néanmoins beaucoup. Les travaux d'Eric Poll [299] proposaient d'inférer la SM de TLS en utilisant LearnLib qui effectue de l'inférence de SM grâce à l'algorithme  $L^*$ <sup>123</sup> [314] lui-même basé

---

123. <https://github.com/LearnLib/learnlib/wiki/Learning-Algorithms-in-LearnLib>

sur une modélisation de machines de Mealy. Le test d'équivalence utilise la méthode de Chow *W-Method*, une méthode d'équivalence approximée.

Certains travaux utilisent la SM inférée pour générer des cas de test à partir des entrées sorties [111]. Malgré tout, inférer et arriver à dériver des cas de test pour un protocole inconnu reste à ce jour imprécis et hasardeux.

#### 9.8.4 Approche en *grey-box*

Le fuzzing d'une cible traitant des informations reçues du réseau ne peut être traité aussi simplement que les programmes recevant l'information par les entrées standard. Cependant, il n'est pas impossible de fuzzer des serveurs web ou FTP. Bien souvent, le fuzzing nécessite un travail préalable sur ces cibles. En effet, les outils de fuzzing usuels disposent rarement de la capacité de communiquer via le réseau (pour des raisons de complexité, mais aussi de performance). Il est possible de modifier le comportement de la cible pour qu'elle obtienne les informations dans des fichiers, plutôt que par une connexion réseau. Les outils de fuzzing standards deviennent alors facilement compatibles. Certains outils tentent de rendre ce processus plus simple, en s'interposant sur les appels relatifs au code réseau, tels que (`socket()`, `bind()`, `listen()` ou `accept()`). L'outil `Preeny`<sup>124</sup> en est un exemple, utilisé notamment pour réaliser du fuzzing réseau avec `AFL`<sup>125</sup>.

On retrouve des exemples de fuzzing dans le dépôt Github d'`Honggfuzz` par exemple. Il met en oeuvre un fuzzing d'`Apache`<sup>126</sup> (un patch d'`Apache` accompagne l'exemple), mais aussi un fuzzing de la pile IP du kernel Linux<sup>127</sup>. L'outil `ffw` [300] propose une approche très intéressante combinant `Radamsa` et `Honggfuzz` avec une surcouche en Python pour du fuzzing de serveurs.

##### Note

On retrouve des éléments intéressants dans les patchs apportés par `Honggfuzz` pour fuzzer la pile IP Linux<sup>a</sup>. Ils modifient par exemple la pile IP pour désactiver les checksums des paquets, et ainsi faciliter le fuzzing, là où certains autres fuzzers tels que `TaintScope`<sup>b</sup> préfèrent une résolution automatique de ces checksums.

- a. [https://github.com/google/honggfuzz/blob/master/examples/linux\\_kernel\\_ip/linux-kernel-4.10.0.patch](https://github.com/google/honggfuzz/blob/master/examples/linux_kernel_ip/linux-kernel-4.10.0.patch)
- b. [http://faculty.cs.tamu.edu/guofei/paper/TaintScope\\_Oakland10\\_slides.pdf](http://faculty.cs.tamu.edu/guofei/paper/TaintScope_Oakland10_slides.pdf)

Le fuzzing en *greybox* appliqué aux programmes réseau permet de conserver les notions de couverture de code et les éventuelles autres fonctionnalités permettant de découvrir des bugs.

124. <https://github.com/zardus/preeny>

125. <https://blog.fuzzing-project.org/27-Network-fuzzing-with-American-fuzzy-lop.html>

126. <https://github.com/google/honggfuzz/tree/master/examples/apache-htpd>

127. [https://github.com/google/honggfuzz/tree/master/examples/linux\\_kernel\\_ip](https://github.com/google/honggfuzz/tree/master/examples/linux_kernel_ip)

### 9.8.5 Conclusion

Les approches les plus prometteuses sont sans conteste les approches *grey-box* tel que `ffw` [300] mais elles impliquent d'avoir accès au programme. Néanmoins aucune d'entre elles ne permet de fuzzer les couches basses Ethernet et IP. Moyennant la redirection par des fichiers, il est aussi possible d'adapter certains fuzzers *grey-box* pour qu'ils supportent le fuzzing réseau. Néanmoins, de par les différents problèmes posés par le fuzzing de protocoles, il est très difficile avec les outils existants de mettre en oeuvre des campagnes de fuzzing performantes et productives.

## 9.9 Outils de fuzzing

Grâce aux différentes ressources examinées [246] dans le cadre de cet état de l'art il a été possible d'établir une liste des outils de fuzzing existants et de leurs capacités. Seuls quelques-uns ont pu être étudiés et évalués sur la suite de test. Pour la plupart des fuzzers un rapide examen a permis d'en extraire les principales caractéristiques regroupées dans le tableau 9.2. Les fuzzers *white-box* ne figurent pas dans ce tableau et se trouvent dans la partie V “Combinaisons d’analyses”. Par ailleurs, de par la grande quantité d’outils publiés, ce tableau ne peut pas être exhaustif et reflète l’état de l’art à la date de la rédaction de ce rapport (*début 2019*). Pour plus de clarté, les caractéristiques non satisfaites par des outils ont été laissées vides.

Certains fuzzers ont intentionnellement été exclus du tableau (notamment [`neuuz`, 353]), car ils se présentent comme des passes de traitement des entrées ou des composants du fuzzing, mais pas un fuzzer complet. D’autres ont aussi été exclus, car ils ont pour seul but de fuzzer un format ou un composant particulier d’un logiciel existant. Parmi ceux-ci, on peut citer `gofuzz`<sup>128</sup> dédié au fuzzing de programmes écrits en Go, `wfuzz`<sup>129</sup> dédié au fuzzing d’application web ou encore `dfuzzer`<sup>130</sup> servant à fuzzer les messages sur le protocole D-Bus. Le chapitre 10 présente et décrit en détail les ressources et les outils sélectionnés pour une analyse approfondie.

## 9.10 Plateformes et infrastructures de fuzzing

Concomitamment au développement des outils et des techniques de fuzzing s'est développé tout un écosystème permettant d'agréger et d'exécuter en parallèle de manière distribuée certains d'entre eux. L'exemple le plus marquant est l'initiative OSS-Fuzz [146] de Google fournissant au grand public une infrastructure de fuzzing en continu de logiciels open source. Cette initiative s'articule autour de ClusterFuzz [145] fournissant l'infrastructure. Ce projet a par ailleurs été depuis le 7 février 2019 complètement open source.

---

128. <https://github.com/Google/gofuzz>

129. <https://github.com/xmendez/wfuzz>

130. <https://github.com/matusmarhefka/dfuzzer>

## 9.10. Plateformes et infrastructures de fuzzing

	Général	Pré trait.	Géné. d'entrées	Exécution de la cible
	Granularité	Instrumentation	Gestion graines	Optimisations
	black-box	Open source	Modèle	Mise à jour
	Code source requis			
AFL [383]				
AFLfast [39]	✓	✓		
AFLGo [40]	✓	✓		
AFLSmart [283]	✓	✓		
AssetFuzzer [220]	✓	✓		
AtomFuzzer [273]	✓	✓		
BFF [348]	✓	✓		
boofuzz [277]	✓	✓		
CalFuzzer [309]	✓	✓		
Choronzon [396]	✓	✓		
classfuzz [76]	✓	✓		
CLSmith [238]	✓	✓		
CollAFL [128]	✓	✓		
DeadLockFuzzer [195]	✓	✓		
Defensics Fuzz [336]	✓	✓		
DELTA [228]	✓	✓		
DIFUZE [88]	✓	✓		
Digttool [272]	✓	✓		
dizzy [118]	✓	✓		
FairFuzz [231]	✓	✓		
ffw [300]	✓	✓		
FLAX [305]	✓	✓		
FOE [349]	✓	✓		
FuzzSim [365]	✓	✓		
GLADE [25]	✓	✓		
GRR [35]	✓	✓		
HawkEye [69]	✓	✓		
Honggfuzz [335]	✓	✓		
IMF [161]	✓	✓		
InsFuzz [391]	✓	✓		
IoTFuzzer [70]	✓	✓		
jsfunfuzz [308]	✓	✓		
kAFL [306]	✓	✓		
LAF-intel [219]	✓	✓		
LangFuzz [176]	✓	✓		
libfuzzer [338]	✓	✓		
MagicFuzzer [60]	✓	✓		
Nautilus [12]	✓	✓		
Nightmare [215]	✓	✓		
Peach [115]	✓	✓		
PerfFuzz [230]	✓	✓		
PeriScope [327]	✓	✓		
pFuzzer [250]	✓	✓		
ProFuzzer [378]	✓	✓		
PULSAR [132]	✓	✓		
RaceFuzzer [310]	✓	✓		
Radamsa [172]	✓	✓		
Sienna Locomotive [34]	✓	✓		
Sidewinder [329]	✓	✓		
Steelix [235]	✓	✓		
Superion [354]	✓	✓		
SLF [377]	✓	✓		
SlowFuzz [280]	✓	✓		
SymFuzz [63]	✓	✓		
syzkaller [352]	✓	✓		
TLS-Attacker [325]	✓	✓		
TriforceAFL [152]	✓	✓		
UnTracer-AFL [262]	✓	✓		
VeriFuzz [79]	✓	✓		
zzuf [174]	✓	✓		

TABLE 9.2 – Comparaison des caractéristiques des outils de Fuzzing

Cette publication marque un tournant dans les infrastructures de fuzzing, car elle permet à n'importe quelle entreprise de le déployer en interne pour fuzzer des logiciels closed-source. Avant cela, une initiative telle que Maxfuzz [83] développé par Coinbase visait une approche similaire pour combiner AFL et GoFuzz dans une architecture basée sur Docker afin de fuzzer des programmes liés à la blockchain. Enfin d'autres initiatives moins ambitieuses et plus faciles à mettre en oeuvre comme Lucky CAT [122] visent à automatiser au maximum une tâche de fuzzing en permettant une gestion aisée des différentes tâches de fuzzing tournant en parallèle. L'utilisation d'une telle infrastructure fournit une réelle plus-value, mais son utilisation dans le cadre du projet irait bien au-delà des exigences du CCTP.

## 9.11 État de l'art : Conclusion intermédiaire

Toutes les étapes d'une campagne de fuzzing ont été détaillées dans les sections précédentes. Celles-ci sont résumées sur la figure 9.1 qui représente les étapes usuelles de fuzzing pour les outils semblables à AFL ou Honggfuzz. Le fuzzing s'avère plus complexe à mettre en oeuvre que le DSE, car il fait intervenir différentes étapes telles que la mutation des entrées, le triage des crashes, etc.

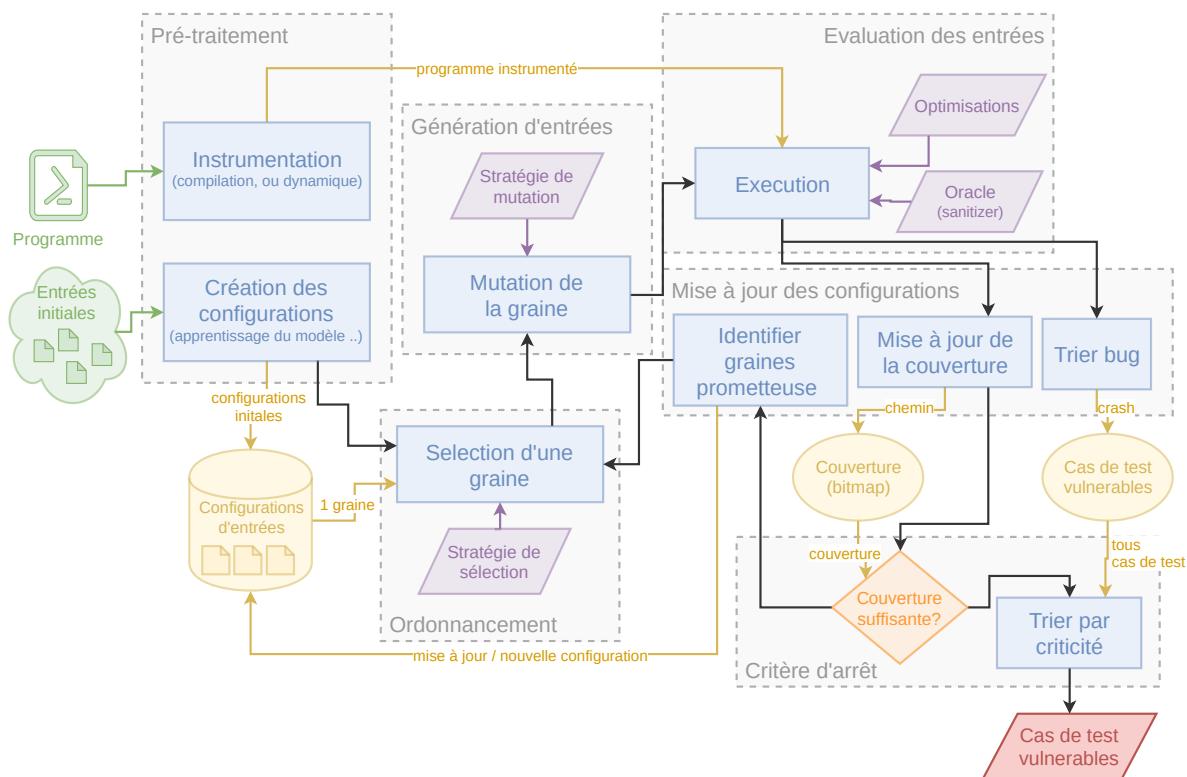


FIGURE 9.1 – Processus de fuzzing

En sus, cet état de l'art met en valeur les différents axes de recherche explorés et les différents outils reflétant l'hétérogénéité des approches pour attaquer le fuzzing. Le tableau 9.3 synthétise les différentes questions de recherches évoquées qui justifient la fécondité de ce domaine de recherche.

Id	Description
RQ_F1	Comment maximiser la fonction objective du fuzzing ? Autrement dit comment prioriser les configurations à exercer et quels sont les critères significatifs d'une configuration prometteuse ( <i>pour de la recherche de vulnérabilités</i> ) ? ( <i>problème FCS</i> )
RQ_F2	Quelles sont les métriques significatives lors de l'exécution sur lesquelles baser une inférence de modèle ?
RQ_F3	Quel ordonnancement des mutations et quelles stratégies de mutation appliquer en fonction du contexte d'exécution ?
RQ_F4	Comment réduire la combinatoire sur l'alphabet d'entrée pour couvrir les séquences d'octets fixes et les entiers 32/64 bits ?
RQ_F5	Comment identifier deux bugs ayant la même origine (la même cause) pour la déduplication de cas de test ?
RQ_F6	Comment trier les cas de test finaux par criticité et exploitabilité ?
RQ_F7	Comment identifier un ensemble minimal de graines représentatives de l'espace des entrées possibles ? ( <i>seed selection problem</i> )
RQ_F8	Quels critères de couvertures utiliser pour maximiser la détection de vulnérabilités ? ( <i>nœud, arc, chemin</i> )

TABLE 9.3 – Récapitulatif des questions de recherche en fuzzing

L'axe de recherche ayant le plus progressé est RQ\_F4, car il s'est avéré un facteur très limitant dans le fuzzing de format de fichiers. Ainsi, plusieurs approches ont été proposées pour répondre à ce problème (cf. section 10.4.2). Toutefois, le fuzzing de manière générale est un domaine de recherche très foisonnant répondant à un besoin toujours plus élevé de sécurité logicielle.



# Chapitre 10

---

## Analyse détaillée des outils

---

### 10.1 Outils de Fuzzing sélectionnés

À partir des données accumulées sur les caractéristiques générales des techniques de fuzzing et des outils de fuzzing (cf. tableaux 9.2), un choix a été effectué pour analyser plusieurs ressources. Le choix s'est naturellement orienté vers les ressources possédant un outil pouvant être testé et satisfaisant les caractéristiques requises par le [Cahier des Clauses Techniques Particulières \(CCTP\)](#). Les avancées récentes étant significatives, le choix s'est particulièrement tourné vers les papiers et les outils publiés récemment.

Le tableau 10.1 résume les caractéristiques détaillées de chaque outil retenu à savoir AFL, Honggfuzz, AFL/QBDI et PULSAR. Une analyse détaillée de chaque outil est effectuée dans la section idoine.

Ce tableau synthétique permet de mettre en lumière les avantages et les inconvénients de chacun d'entre eux pour les comparer. Différents critères ont été sélectionnés afin de donner au lecteur une vue d'ensemble des outils. Ils permettent notamment de situer la maturité des outils, leur support pour les architectures, les plateformes et les fonctionnalités modernes. A noter que le support de ARM de PULSAR est indépendant de l'outil puisqu'il fonctionne en *black-box*. Afin de compléter cette vision synthétique et superficielle, chacun des outils a été analysé séparément et en détail dans les sections 10.2, 10.3, 10.4 et 10.5.

			#1 : AFL	#2 Honggfuzz	#3 AFL/QBDI	#4 PULSAR	
CT0	CT0_1 : Langage	x86	✓	✓	✓	✓	
	CT0_2 : Architecture	x86-64	✓	✓	✓	✓	
		ARMv7	✗	✓	~	✓	
		ARMv8	✗	✓	~	✓	
	CT0_3 : Open source		✓	✓	✓	✓	
	CT0_4 : Licence	Apache 2	Apache 2	Apache 2	BSD 3		
	CT0_5 : Documentation		~	~	✓	✗	
CT0_F	CT0_6 : Activité		~	✓	✓	✗	
	CT0_7 : Jeu de tests		✓	✓	✓	✗	
	CT0_F1 : Granularité		●	●	●	●	
	CT0_F2 : Code requis		✓	✓	✗	✓	
	CT0_F3 : Instrumentation	statique	✓	✓	✓	✓	
		dynamique	✗	✗	✓	✗	
	CT0_F4 : Ordonnancement		✓	✓	✓	✗	
Gestion des entrées	CT0_F5 : Modèle	grammaire	✗	✗	✗	✗	
		inféré	✗	✗	✗	✓	
	CT0_F6 : Mutation	aléatoire	✓	✓	✓	✓	
		intelligente	✓	✓	✓	✓	
	CT0_F7 : Appels systèmes		✗	✗	✗	✗	
	CT0_F8 : Fuzzing réseau		✗	✓	✗	✓	
	CT0_F9 : Optimisation	serveur Fork	✓	✗	✓	✗	
Exécution de la cible		fuzzing en mémoire	✓	✗	✓	✗	
CT0_F10 : Sanitizer	spatial		✓	✓	✓	✗	
	temporel		✓	✓	✓	✗	
	mauvaise coercition		✗	✗	✗	✗	
	comporte. indéfinis		✓	✓	✓	✗	
	thread		~	✗	~	✗	
	test différentiel		✗	✗	✗	✗	
	CT0_F11 : Triage de crashes	déduplication	✓	✓	✓	✗	
CT0_C		priorisation	✓	✓	✓	✗	
		génétique	✓	✓	✓	✗	
		réduction	✓	✓	✓	✗	
CT0_C12 : Mise à jour de C							

TABLE 10.1 – Comparaison des outils sélectionnés

## 10.2 Fuzzing #1 : American Fuzzy Lop (AFL)

American Fuzzy Lop (AFL) est un fuzzer développé par Michal Zalewski sous la licence *Apache Licence 2.0*. Il a été activement maintenu depuis 2014 jusqu'en 2017 et dispose d'une communauté d'utilisateurs active via le groupe Google *afl-users*. La dernière mise à jour d'AFL date de novembre 2017 et correspond à la version 2.52b. Le code disponible sur le site de l'auteur<sup>131</sup> n'est pas versionné par un gestionnaire de version (Git, SVN).

```

american fuzzy lop 2.52b (demo)

process timing
  run time : 0 days, 0 hrs, 0 min, 9 sec
  last new path : none yet (odd, check syntax!)
  last uniq crash : none seen yet
  last uniq hang : none seen yet
cycle progress
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 96/256 (37.50%)
  total execs : 63.7k
  exec speed : 4869/sec
fuzzing strategy yields
  bit flips : 0/32, 0/31, 0/29
  byte flips : 0/4, 0/3, 0/1
  arithmetics : 0/223, 0/107, 0/4
  known ints : 0/18, 0/69, 0/44
  dictionary : 0/0, 0/0, 0/0
  havoc : 0/63.0k, 0/0
  trim : 99.96%/17, 0.00%
overall results
  cycles done : 243
  total paths : 1
  uniq crashes : 0
  uniq hangs : 0
map coverage
  map density : 0.01% / 0.01%
  count coverage : 1.00 bits/tuple
findings in depth
  favored paths : 1 (100.00%)
  new edges on : 1 (100.00%)
  total crashes : 0 (0 unique)
  total tmouts : 0 (0 unique)
path geometry
  levels : 1
  pending : 0
  pend fav : 0
  own finds : 0
  imported : n/a
  stability : 100.00%
[cpu000: 34%]
```

FIGURE 10.1 – Interface utilisateur - AFL

AFL se présente sous la forme de plusieurs exécutables. Le plus notable, **afl-fuzz**, implémente le fuzzer en tant que tel. Il est accompagné des compilateurs **afl-gcc** et **afl-g++** qui servent à instrumenter les programmes cibles. Enfin, divers outils accompagnent le fuzzer. Par exemple, **afl-tmin** et **afl-cmin** permettent une optimisation du corpus de test, d'autres outils tels que **afl-analyse** peuvent être utilisés avec **afl-fuzz** pour rendre le fuzzing plus efficace.

Le code source est accessible dans un format assez compact. En effet, à titre d'exemple, le code source du fuzzer **afl-fuzz** est contenu dans un unique fichier C de plus de 8000 lignes, rendant la compréhension et la prise en main de l'outil difficile, même pour les utilisateurs les plus avancés. Le manque de flexibilité des sources d'AFL rend peu pratiques

131. <http://lcamtuf.coredump.cx/afl/>

les améliorations ou les modifications apportées.

AFL n'est pas accompagné d'un papier académique ou d'une documentation étouffée. Cependant, plusieurs documents disponibles sur le site de l'auteur [383] expliquent comment utiliser le fuzzer et présentent certains détails d'implémentation (en particulier [387]).

### 10.2.1 Prétraitement

#### Instrumentation

AFL instrumente le programme cible afin de récolter des informations sur la couverture de code. Cette instrumentation est possible grâce aux compilateurs alternatifs fournis (`afl-gcc` et `afl-g++`). L'instrumentation peut aussi être dynamique, par l'intermédiaire de l'outil QEMU [26], et permet de fuzzer une cible sans disposer de ses sources. L'instrumentation statique reste cependant recommandée, car bien plus performante que l'instrumentation se basant sur QEMU. L'instrumentation proposée par AFL est relativement simple et est équivalente au code présenté dans Listing 23.

---

#### Listing 23 Code d'instrumentation d'AFL - bloomfilter

---

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

---

Le principe derrière ces quelques lignes de code repose sur le mécanisme du `bloom-filter`. Cette structure de données est particulièrement efficace dans le cadre du fuzzing et de la couverture de code : son accès y est rapide, les faux positifs restent peu nombreux, permettant ainsi d'obtenir un bon compromis entre vitesse d'exécution et exactitude des résultats.

Le `bloom-filter`, symbolisé par la variable `shared_mem`, est un espace mémoire partagé entre le fuzzer et la cible, permettant au fuzzer d'obtenir des informations sur la couverture du code du programme fuzzé.

#### Note

La taille de cet espace mémoire est définie à 64 Ko. Cette taille a été définie à l'époque de manière à ce que le `bloom-filter` soit adapté à la taille du cache L2 du CPU. Cette taille peut donc varier, pour optimiser le ratio vitesse d'exécution/faux positifs, en prenant en compte l'espace disponible dans le cache L2.

Le `bloom-filter` est mis à jour à chaque branche grâce au code d'instrumentation d'AFL. À chaque branche, un nombre aléatoire (`COMPILE_TIME_RANDOM`) permet de déterminer

l'emplacement actuel du fuzzer. Il est “xorré” avec la localisation précédente dans le fuzzer divisée par 2 (`prev_location`). Cette opération permet non seulement au fuzzer de déterminer sa position, mais aussi le chemin emprunté pour y arriver. La variable `prev_location` est alors mise à jour avec la valeur de la localisation actuelle du fuzzer en vue de la prochaine branche.

#### Note

La mémoire partagée `shared_mem` est mise à jour à chaque nouvelle branche. On retrouve dans cette variable le nombre de fois où cet emplacement a été atteint en empruntant un chemin spécifique. Ce nombre permet d'informer le fuzzer sur l'intérêt d'un chemin via un système de buckets. Ce processus est décrit plus en détail dans la section 2) *Detecting new behaviors* du document [387].

## Optimisations

**Fork server** Durant une campagne de fuzzing, à chaque lancement du programme cible, du temps est consommé pour son chargement et l'édition de liens. Ce temps d'initialisation impacte fortement les performances du fuzzer. Afin de maximiser le temps effectif du fuzzer, AFL utilise une méthode qu'il nomme *fork-server* [385]. Cette astuce permet de fuzzer le programme cible de manière répétée sans passer par la phase d'initialisation à chaque nouveau processus. Le fork-server est une boucle d'attente ajoutée dans le programme cible avant le point, depuis lequel on veut commencer à fuzzer. Bien souvent, cette boucle prend place une fois le chargement initial du programme effectué. Le forkserver est capable de communiquer avec `afl-fuzz` et attend qu'AFL l'autorise à continuer l'exécution du programme cible. Quand l'exécution est ordonnée, le fork-server utilise l'appel système `fork()` qui a pour conséquence de diviser le programme cible en deux processus :

- Processus fils : il correspond à une nouvelle instance du programme cible qui s'exécute normalement et devient la cible du fuzzer. Le mécanisme de *copy-on-write* permet de créer cette nouvelle instance sans consommer excessivement les ressources disponibles ;
- Processus parent : il transmet au fuzzer le **PID** du processus fils, et retourne dans la boucle d'attente du fork-server jusqu'à ce qu'une nouvelle exécution lui soit commandée par AFL.

**Parallélisation** Chaque instance d'`afl-fuzz` utilise un unique cœur du processeur. Pour améliorer les performances sur un système multicœurs il est possible de lancer plusieurs instances d'AFL en parallèle [388]. Les différentes instances s'exécutent de façon indépendante, mais se synchronisent régulièrement afin d'échanger des entrées intéressantes entre elles. AFL permet également de synchroniser des instances d'`afl-fuzz` sur des systèmes distants via SSH.

**Mode persistant** AFL dispose d'un mode persistant qui s'apparente à du *in-memory fuzzing*. Le mode persistant consiste à instrumenter le programme cible de façon à pouvoir le faire boucler et faire s'exécuter la même instance (sans véritable réinitialisation) successivement sur des entrées différentes [386]. Cela permet un gain de temps considérable, mais induit le risque de détecter de nombreux bugs qui seront des faux positifs si le programme boucle depuis un état qui n'est pas un état initial valide. L'instrumentation correspondant au mode persistant doit être faite manuellement dans les sources du programme cible.

### 10.2.2 Gestion des entrées

`afl-fuzz` nécessite d'être lancé avec un répertoire de fichiers (graines) qu'il va utiliser comme base pour générer de nouvelles entrées pour le programme cible. Si aucun fichier d'exemple n'est disponible, l'outil `afl-analyse` permet d'inférer automatiquement le format d'entrée attendu par le programme cible.

Durant le fuzzing d'un programme, AFL maintient une queue d'entrées en attente. Cette queue est initialisée avec les graines initiales données en argument au fuzzer. À chaque nouvelle itération du fuzzer, une entrée est sélectionnée et utilisée pour générer de nouvelles entrées par mutation. Si ces nouvelles entrées sont jugées intéressantes (*i.e* si elles empruntent de nouvelles branches ou si elles générèrent la même couverture en étant plus rapides ou plus petites que les anciennes entrées) alors elles sont ajoutées en priorité à la queue d'entrées.

L'exécution d'une itération complète de fuzzing (mutations et exécutions de la cible) est implémentée au sein de la fonction : `static u8 fuzz_one(char** argv)`. Différentes mutations sont appliquées par AFL :

- *inversion de bits* : inversion aléatoire de bits
- *mutation arithmétique* : l'entrée est interprétée comme une suite d'entiers qui sont modifiés selon une équation arithmétique. AFL effectue des mutations par groupes de 1, 2, ou 4 octets.
- *valeurs particulières* : insertion dans l'entrée de valeurs qui ont souvent une signification particulière en informatique, par exemple 0, 1, `MAX_INT`, etc.
- *dictionnaire*[381] : la mutation par dictionnaire fonctionne sur le même principe que la mutation par valeurs intéressantes, à la différence près que les valeurs du dictionnaire sont ajoutées de deux façons :
  - par l'utilisateur dans un fichier ou un répertoire
  - par AFL automatiquement. Lorsqu'AFL détecte dans une entrée une séquence d'octets qui, dès que mutée, change radicalement la couverture de code, mais dont les octets avoisinants n'ont pas d'influence sur la couverture, alors cette valeur est ajoutée au dictionnaire.
- *modifications aléatoires* : il s'agit de l'application aléatoire des différentes stratégies de mutation décrites précédemment sur l'entrée.

- *splicing* : deux entrées sont mélangées pour produire une nouvelle entrée, qui est ensuite modifiée par la mutation *modifications aléatoires*

### 10.2.3 Exécution de la cible

#### Détection de bugs

Par défaut, AFL détecte des bugs en vérifiant si le programme cible s'est terminé normalement ou s'il a été terminé par un signal d'erreur. Il peut également être utilisé en combinaison avec des *sanitizers* afin de détecter des bugs qui ne font pas crasher le programme cible. AFL supporte deux *sanitizers*[382] :

- **AddressSanitizer (asan)** [312] : il permet de détecter de nombreuses erreurs mémoire comme les *use-after-free*, *buffer-overflow*, *use-after-scope*, fuites mémoire, etc ;
- **MemoryMSanitizer (msan)** [332] : il permet de détecter les lectures à des emplacements mémoire non initialisés.

#### Dislocator

Une alternative à l'utilisation d'ASAN[312] est une bibliothèque fournie par AFL nommée Dislocator. Elle est moins précise qu'un *sanitizer* à proprement parler (pas de détection des dépassements sur la pile), mais permet d'être plus rapide et performant sur des exécutables de petite taille par exemple. De plus, cette bibliothèque se chargeant à l'aide de `LD_PRELOAD` ou `DYLD_INSERT_LIBRARIES`, permet facilement de s'interposer lors d'un fuzzing en source fermée.

#### Note

La bibliothèque Dislocator a servi de base au développement d'un module d'interposition dans AFL/QBDI, présenté dans la section 10.4.2.

#### Choix des entrées

AFL Possède un mécanisme pour marquer certaines entrées générées comme *prioritaires* ( le mot-clé "favored" est utilisé dans l'implémentation et la documentation d'AFL). Lorsqu'une nouvelle itération de fuzzing débute, les entrées prioritaires seront préférées comme base pour la génération de nouvelles entrées. La priorisation des entrées jugées les plus intéressantes a un objectif triple :

- Réduire "*virtuellement*" la taille de la queue d'entrées : pour réduire le nombre d'entrées potentielles, AFL sélectionne les entrées dont la couverture de code inclue celle d'autres entrées, et donnera la priorité aux entrées qui "englobent" les autres.

L'idée est de couvrir le maximum de branches avec une unique exécution. Il est à noter que les entrées "non-prioritaires" sont toujours conservées dans la queue d'entrées possibles, bien qu'elles soient utilisées bien plus rarement que les entrées prioritaires ;

- *Explorer intelligemment l'espace possible des entrées* : après une itération de fuzzing, AFL sélectionne les nouvelles entrées qui ont généré un nouveau chemin d'exécution et les ajoute en position prioritaire dans la queue d'entrées possibles. L'objectif est de permettre une exploration des différents formats possibles pour les entrées ;
- *Optimiser les performances* : AFL classe les entrées selon un score calculé par  $\text{temps\_execution} \times \text{taille}$ . La priorité est ensuite donnée aux entrées dont le score est minimal (c'est à dire les plus petites et les plus rapides).

En pratique, cette heuristique de priorisation des entrées est implémentée grâce à deux fonctions :

```
static void cull_queue(void);
static void update_bitmap_score(struct queue_entry* q);
```

Et deux variables globales :

```
EXP_ST u8* trace_bits;
static struct queue_entry* top_rated[MAP_SIZE];
```

Leurs rôles sont décrits ci-dessous :

- **trace\_bits** est une version optimisée de **shared\_mem[]** qui représente les branches empruntées sans considérer le nombre de passages ;
- **top\_rated** est un tableau qui donne, pour chaque branche du programme cible, l'entrée qui couvre cette branche et dont le score est minimum ;
- **update\_bitmap\_score()** est la fonction qui maintient **top\_rated** à jour. Elle vérifie pour chaque branche du programme si l'entrée donnée en argument (*q*) l'emprunte en ayant un meilleur score que la précédente entrée marquée comme **top\_rated** pour cette branche. Si c'est le cas, *q* devient la nouvelle **top\_rated** de la branche ;
- **cull\_queue** est la fonction qui détecte si de nouvelles branches ont été découvertes après un round de fuzzing. Si une nouvelle branche est détectée alors l'entrée **top\_rated** (i.e l'entrée la plus petite et la plus rapide qui emprunte cette branche ) de la branche est marquée comme prioritaire.

Au terme de chaque round de fuzzing, **update\_bitmap\_score()** est appelée pour mettre à jour la liste **top\_rated**. Ensuite **cull\_queue** est appelée et utilise **trace\_bits** et **top\_rated** pour choisir les entrées qui sont à marquer comme prioritaires.

## Déduplication

Afin de rendre les résultats de la campagne de fuzzing facilement exploitables, AFL tente d'identifier et de supprimer les doublons afin de garder les crashes uniques. Pour ce faire, dès qu'un crash est rencontré, AFL utilise une heuristique pour déterminer si le crash est nouveau (auquel cas il est ajouté) ou s'il a déjà été vu auparavant (auquel cas il est ignoré). Un crash est considéré comme nouveau si :

- il a été généré avec une couverture de code qui inclut de nouvelles branches ;
- il a été généré avec une couverture de code dont les branches empruntées sont un sous-ensemble des branches empruntées par les précédents crashes.

Cette heuristique de déduplication est très similaire à l'heuristique utilisée par AFL pour sélectionner les entrées prioritaires.

## Minimisation des entrées

AFL tente de réduire la taille des entrées qu'il utilise. Si une entrée plus petite, créée en tronquant une entrée de taille supérieure, génère toujours la même couverture de branches, alors l'entrée la plus conséquente peut être remplacée par la plus petite. L'objectif est de supprimer les parties "inutiles" des entrées du point de vue du fuzzing (*i.e.* qui n'ont pas d'impact sur la couverture de branches). Il minimise les graines au début de la campagne de fuzzing, et minimise également les nouvelles entrées qui sont ajoutées à la queue au fur et à mesure de l'exécution du fuzzer. La fonction qui réduit la taille des entrées est la suivante :

```
static u8 trim_case(char** argv, struct queue_entry* q, u8* in_buf);
```

Elle est appelée à chaque nouvelle itération du fuzzer dans la fonction `fuzz_one()`. `trim_case()` est implémentée de façon non optimale : elle ne cherche pas à minimiser les entrées au maximum, mais simplement à les réduire en trouvant un compromis entre le facteur de réduction de l'entrée et le temps de traitement nécessaire. Si l'on désire minimiser les entrées de façon plus optimale, l'outil `afl-tmin` peut être utilisé. Il implémente des méthodes de réduction de la taille des entrées plus complètes que la fonction `trim_case()`, mais aussi plus coûteuses. À noter que les entrées qui provoquent un crash ne sont pas minimisées pour pouvoir être comparées facilement avec les entrées "valides" depuis lesquelles elles ont été dérivées.

### 10.2.4 Conclusion

AFL est depuis quelques années déjà, l'outil de référence dans le domaine du fuzzing. En effet il était à sa sortie innovant et révolutionnaire par ses approches intelligentes, en se basant sur la couverture de code et sur le retour d'informations vis-à-vis de la cible. De nombreuses recherches se servent d'AFL comme base (de nombreux forks du

projet existent, apportant chacun leurs fonctionnalités). Son passage à l'échelle et sa prise en main rapide facilitent grandement les preuves de concepts pour étayer des travaux de recherche. Néanmoins, il n'est pas exempt de défauts [233] et il n'a pas reçu de mise à jour depuis 2017, et semble à l'abandon. Son manque de documentation précise et la qualité du code du projet discutable rendent toute modification particulièrement laborieuse. Ces différents points semblent être problématiques pour le projet PASTIS et ne font pas d'AFL un candidat assurant la pérennité du projet.

## 10.3 Fuzzing #2 : Honggfuzz

Honggfuzz est un outil de fuzzing développé principalement par Robert Swiecki (Google Zurich) sous la licence *Apache Licence 2.0*. Ce fuzzer a attiré notre attention par son utilisation importante dans OSS-Fuzz [146] pour du fuzzing à grande échelle. De plus, il semble présenter des capacités à fuzzer du code de serveurs web<sup>132</sup>. Le passage à l'échelle de l'outil sur des outils réseau en fait un produit de choix pour cette étude.

```
-----[ 0 days 00 hrs 00 mins 16 secs ]-----
Iterations : 14092 [14.09k]
Mode [2/2] : Feedback Driven Mode
  Target : ./bins/qfuzz-HF-instr
  Threads : 1, CPUs: 4, CPU%: 122% [30%/CPU]
  Speed : 910/sec [avg: 880]
  Crashes : 0 [unique: 0, blacklist: 0, verified: 0]
  Timeouts : 0 [10 sec]
Corpus Size : 14, max: 8192 bytes, init: 0 files
Cov Update : 0 days 00 hrs 00 mins 01 secs ago
Coverage : edge: 13 pc: 0 cmp: 189
----- [ LOGS ] ----- / honggfuzz 1.8 /-
```

FIGURE 10.2 – Interface utilisateur - Honggfuzz

En ce qui concerne l'activité du projet, le premier commit de l'outil date d'octobre 2010, cependant son activité, restée disparate durant les premières années, a été bien plus importante ces dernières années, comme en atteste la figure 10.3



FIGURE 10.3 – Activité de développement - Honggfuzz

132. <https://github.com/google/honggfuzz/blob/master/docs/AttachingToPid.md>

La dernière release d’**Honggfuzz** est la version 1.7 datant d’août 2018. **Honggfuzz** est un fuzzer facilement utilisable et configurable. Il a son actif des CVE dans de nombreux projets : Openssl, Apache, Flash, libtiff, IDA Pro, etc.

Le code du fuzzer est écrit en C, l’organisation du projet est claire et précise. L’arborescence permet facilement d’identifier les différentes parties du fuzzer. Cependant, la documentation est extrêmement réduite. De plus, il n’existe aucun papier expliquant les choix techniques ou les techniques employées dans le fuzzer. Certaines fonctionnalités font d’**Honggfuzz** un fuzzer de choix. En effet, il dispose de modes *multi-process* et *multi-threaded*, permettant en conjonction du mode *persistant*, des performances remarquables.

De plus **Honggfuzz** supporte nativement de nombreuses plateformes : GNU/Linux, FreeBSD, NetBSD, Mac OS X, Windows/CygWin et Android, et cela, sous différentes architectures : x86, x86\_64, ARM32 et ARM64. Dans le cas des CPU Intel, **Honggfuzz** dispose aussi d’un mode de couverture de code reposant sur les technologies matérielles **Intel BTS** (cf. 17.4.5 manuel Intel [186]) et **Intel PT** [187]. Il dispose aussi de la capacité à fuzzer des logiciels reposant sur le réseau, tels que des serveurs web (Apache).

#### Note

Une présentation récente lors de la conférence Area41 a présenté un cas d’usage complet de recherche de vulnérabilités avec **Honggfuzz** combiné avec une surcouche Python [300] et appliqué à des protocoles réseaux, en particulier Message Queuing Telemetry Transport (MQTT).

### 10.3.1 Prétraitement

De manière similaire à de nombreux autres fuzzers, **Honggfuzz** fournit des compilateurs permettant d’injecter du code d’instrumentation lors de la compilation dans la cible. Deux compilateurs par langage (C et C++) sont fournis : **hfuzz-clang**, et **hfuzz-gcc** pour le C, **hfuzz-clang++**, et **hfuzz-g++** pour le C++.

#### Note

Ces quatre compilateurs ne sont en fait que des liens symboliques vers un unique exécutable **hfuzz**. Celui-ci se base sur les installations présentes du système de clang et gcc pour compiler les cibles de fuzzing. Ainsi, les fonctionnalités proposées par **hfuzz** dépendent des versions présentes sur le système.

A l’étape de compilation, le code d’instrumentation injecté ne se présente que sous la forme de callbacks. **Honggfuzz** insère par la suite dans ces callbacks des appels vers la bibliothèque tierce, **libhfuzz**, où se trouve l’intelligence propre au fuzzer.

## libhfuzz

La bibliothèque `libhfuzz` est la charge utile de l'instrumentation d'`Honggfuzz`. Elle fonctionne grâce aux mécanismes de callbacks mis en place lors de la compilation. Libhfuzz propose différentes fonctions de callbacks permettant au fuzzer de compter les évènements à l'aide de bitmaps.

Par exemple, lorsque le mode `-fsanitize-coverage=trace-pc` est activé, le compilateur insère une fonction de callback `__sanitizer_cov_trace_pc()` sur chacune des branches afin de mettre à jour le bitmap relatif à l'évènement déclenché. Les callbacks disponibles sont nombreux et répertoriés dans le fichier source<sup>133</sup>. Chacun d'entre eux permet d'alimenter différentes bitmaps et peut être consulté dans la structure `feedback_t`, présentée dans le listing 24.

---

**Listing 24** Structure `feedback_t`

---

```
typedef struct {
    bool pcGuardMap[_HF_PC_GUARD_MAX];
    uint8_t bbMapPc[_HF_PERF_BITMAP_SIZE_16M];
    uint8_t bbMapCmp[_HF_PERF_BITMAP_SIZE_16M];
    uint64_t pidFeedbackPc[_HF_THREAD_MAX];
    uint64_t pidFeedbackEdge[_HF_THREAD_MAX];
    uint64_t pidFeedbackCmp[_HF_THREAD_MAX];
} feedback_t;
```

---

Le remplissage de cette structure permet au fuzzer de disposer d'une introspection fine et précise du programme et d'utiliser la couverture du programme afin d'orienter le fuzzer vers des résultats plus probants.

### 10.3.2 Gestion des entrées

`Honggfuzz` propose différentes méthodes de gestion des entrées, avec pour objectif de déterminer et de minimiser les entrées. Cette gestion permet toutefois de conserver une couverture maximale du code et des branches grâce au corpus optimisé par `Honggfuzz`. Il existe deux phases de gestion des entrées dans l'outil.

Dans un premier temps, lors du lancement de la campagne de fuzzing, `Honggfuzz` itère sur chaque fichier du corpus initial. Les entrées permettant d'augmenter la couverture (de code, ou de branches) sont ajoutées à une liste d'entrées dynamique. C'est cette liste initiale et optimisée qui sera ensuite utilisée.

Dans un second temps, et tout au long de la campagne, le fuzzer sélectionnera aléatoirement une des entrées du corpus dynamique et mute cette entrée en utilisant diverses

---

133. <https://github.com/google/honggfuzz/blob/master/libhfuzz/instrument.c>

stratégies dont le nom est explicite (et pour beaucoup, communes aux mutations rencontrées dans d'autres fuzzers, tels qu'afl) :

- `mangle_Resize` : modifie la taille de l'entrée ;
- `mangle_Byt`e : remplace un entier de 64 bits ;
- `mangle_Bit` : modifie un bit d'un octet ;
- `mangle_Byt`es : modifie des entiers de 16, 32 ou 64 bits ;
- `mangle_Magic` : modifie un entier de 8, 16, 32 ou 64 bits par des valeurs arbitraires ;
- `mangle_IncByt`e : incrémente un entier ;
- `mangle_DecByt`e : décrémente un entier ;
- `mangle_NegByt`e : complément à 1 d'un entier ;
- `mangle_AddSub` : ajoute ou soustrait une valeur ;
- `mangle_Dictionar`y : remplace une valeur par un mot du dictionnaire ;
- `mangle_Dictionar`y`Insert` : ajoute un mot du dictionnaire ;
- `mangle_MemMove` : remplace une partie de l'entrée ailleurs dans l'ensemble ;
- `mangle_MemSet` : initialise une partie de l'entrée à une valeur fixe ;
- `mangle_Random` : modifie un entier de 16, 32 ou 64 bits de manière aléatoire ;
- `mangle_CloneByt`e : copie un octet vers un autre emplacement.

Note

Cette liste n'est pas exhaustive. Beaucoup de ces mutations proposent une variante *printable*. En prenant l'exemple de `mangle_IncByt`e, la version *printable* `mangle_IncByt`e`Printable` permettra de modifier la valeur de l'octet, tout en s'assurant qu'il reste dans l'espace des caractères affichables au sens de la table ASCII.

Une fois la mutation appliquée, le fuzzer vérifie si elle étend la couverture actuelle du programme. Dans le cas où de nouvelles branches ou de nouvelles instructions sont découvertes, l'entrée mutée est ajoutée au corpus dynamique (et stockée sur le disque).

Note

Plusieurs mutations peuvent être appliquées à la fois. `Honggfuzz` permet de paramétrier ces mutations lors du lancement du fuzzer.

Enfin, une particularité notable d'`Honggfuzz` est sa modularité vis-à-vis du traitement des entrées. En effet, il dispose d'une option permettant à l'utilisateur de définir une source de mutation externe à l'aide du paramètre `-mutate-cmd`.

### 10.3.3 Exécution de la cible

La boucle de fuzzing de `Honggfuzz` est relativement simple et se définit en 4 étapes :

1. Réinitialiser les variables (compteurs, bitmaps) ;
2. Récupérer et muter une entrée ;
3. Exécuter la cible à l'aide de la nouvelle entrée ;

4. À l'aide du retour apporté par les bitmaps, ou un éventuel crash, décider si l'entrée mutée doit être ajoutée au corpus dynamique ;
5. Nouvelle itération de la boucle de fuzzing.

Malgré la boucle de fuzzing qui peut paraître simpliste, les résultats générés par Honggfuzz restent intéressants, grâce aux optimisations que présentent ce fuzzer.

## Optimisations

**Persistance** Une optimisation majeure de Honggfuzz est la persistance en mémoire. Ce mécanisme permet de réutiliser le contexte mémoire d'un processus plusieurs fois. En effet, la création d'un processus est relativement coûteuse en temps, la réutilisation d'un processus déjà existant permet d'améliorer nettement les performances du fuzzer. L'activation de la persistance est particulièrement intéressante lors du fuzzing d'API. Deux méthodes sont proposées pour fuzzer un programme en faisant usage de la persistance : le mode *ASAN-style* et *HF\_ITER-style*, tous deux décrits dans la documentation<sup>134</sup>.

**Multi-threading et multi-process** Honggfuzz permet d'utiliser les capacités de la machine de fuzzing à son plein potentiel grâce au multi-threading et au multi-processing qu'il propose. En outre, ces fonctionnalités permettent de distribuer le corpus de fuzzing de manière intelligente aux différents processus et threads créés.

**Couverture de code matérielle** Un des points forts d'Honggfuzz est sa capacité à prendre avantage des récentes technologies hardware telles que **Intel PT**<sup>135</sup> et **Intel BTS**<sup>136</sup> sur les architectures x86 et x86\_64. Ces technologies modernes permettent d'utiliser des spécificités matérielles des processeurs pour obtenir des informations sur la couverture de code, le flot d'exécution du programme, mais aussi des informations sur les interruptions et les exceptions. La couverture de code matérielle réalisée par Honggfuzz à l'aide de ces technologies se combine aisément avec le rétrocontrôle logiciel. Cela permet d'obtenir une couverture de code rapide, tout en conservant l'intelligence logicielle sur la gestion et l'analyse des entrées. On obtient grâce à cette fonctionnalité des performances remarquables.

## Cassage de comparaisons

Lors de la compilation à l'aide d'`hfuzz`, il existe la possibilité d'ajouter du code d'instrumentation au niveau des instructions de comparaisons. Le fuzzer est alors à même de

---

134. <https://github.com/google/honggfuzz/blob/master/docs/PersistentFuzzing.md>

135. <https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing>

136. <https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-bts.txt>

casser la comparaison efficacement (voir section 10.4.2). Ici, Honggfuzz propose au travers des méthodes : `__sanitizer_cov_trace_cmpX`<sup>137</sup> une découverte efficace de l'entier attendu. Pour cela, il compare les bits similaires entre l'entrée et la valeur attendue. Si le nombre de bits similaires augmente, l'entrée sera conservée, dans le cas contraire, une nouvelle itération est effectuée avec une mutation d'autres bits. On répète l'opération jusqu'à obtenir la valeur attendue, et ainsi casser les comparaisons, à priori onéreuses pour un fuzzer simpliste.

### 10.3.4 Gestion des crashes

**Détection des crashes.** Honggfuzz utilise plusieurs méthodes afin de détecter un éventuel crash. La première, que l'on retrouve dans de nombreux autres outils, est l'analyse des signaux générés par le programme, afin de déterminer s'il ne s'est pas terminé correctement. Une seconde méthode proposée par Honggfuzz est l'utilisation de `ptrace`. Cette méthode est réservée aux plateformes Linux et NetBSD et permet de découvrir les signaux qui auraient été ignorés, ou interceptés par la cible, ne provoquant donc pas un crash du programme.

**Assainisseur d'adresses.** Certains types de bugs ne génèrent pas de crashes, mais peuvent cependant présenter un intérêt pour l'utilisateur. Ces erreurs silencieuses peuvent être détectées par Honggfuzz à l'aide d'un assainisseur d'adresses (cf. Section 9.6.2). Il en propose un certain nombre dont *AddressSanitizer* [312], *UBSan* [103], *ThreadSanitizer* [313] ou *LeakSanitizer* [224]. Cette fonctionnalité a un coût sur les performances du fuzzer, mais permet d'analyser plus en profondeur le programme, et de détecter des bugs qui auraient pu passer inaperçu.

**Déduplication** Honggfuzz permet d'analyser les crashes trouvés lors de la campagne de fuzzing et de détecter les crashes similaires. Cette détection est possible à l'aide de la bibliothèque tierce `libunwind` [360]. La dé-duplication des crashes se base sur l'historique de la pile d'appels. Un hash est effectué sur cet historique et sert de référence lors de la comparaison de crashes.

### 10.3.5 Conclusion

Honggfuzz semble être un outil prometteur. En effet, son utilisation par le projet OSS-Fuzz et les résultats générés attestent de sa scalabilité et de son efficacité. De plus, l'état actuel de la base de code est modulaire et flexible, et semble permettre des modifications aisées. Ses différentes optimisations, et sa capacité à fuzzer des serveurs web assurent une certaine confiance sur les résultats pouvant être obtenus par ce fuzzer. Un autre des critères non négligeables du projet est aussi sa capacité à supporter de multiples plateformes

---

137. <https://github.com/google/honggfuzz/blob/master/libhfuzz/instrument.c#L112>

### *10.3. Fuzzing #2 : Honggfuzz*

---

et architectures. Plus particulièrement, son support natif pour l'ARM32 et l'ARM64 renforce l'intérêt pour le projet. [Honggfuzz](#) semble particulièrement adapté au projet et les différents points cités précédemment en font un sérieux candidat pour le projet [PASTIS](#).

## 10.4 Fuzzing #3 : AFL/QBDI

AFL/QBDI est un fuzzer basé sur AFL et sur QBDI développé par Quarkslab. Il a la particularité et l'intérêt de pouvoir fuzzzer des exécutables compilés, sans nécessiter l'option `-Q` (QEMU) d'AFL, permettant ainsi d'atteindre des performances supérieures. Ainsi, dans la majorité des cas, AFL/QBDI se comporte comme AFL, mais sur des exécutables dont les sources ne sont pas disponibles. En l'état, aucune modification n'est effectuée dans AFL, AFL/QBDI se déclinant comme une variante du mode QEMU. Ainsi, la version standard d'AFL peut être utilisée sans modifications de son code source. (A l'exception de l'optimisation décrite dans la section 10.4.2)

AFL/QBDI fonctionne aujourd'hui sur les plateformes `UNIX` dans l'architecture `x86_64`. Son fonctionnement sur d'autres architectures dépend directement des architectures supportées par QBDI. L'utilisation d'un outil d'instrumentation dynamique, plutôt que l'instrumentation statique proposée par AFL, permet d'implémenter différentes optimisations et de rendre le fuzzer bien plus efficace que la version originale de l'outil. Autrement, l'essentiel des mécanismes de fuzzing sont ceux d'AFL décrits en Section 10.2. Seront principalement décrits dans ce qui suit le fonctionnement de l'outils et les optimisations apportées.

### 10.4.1 Exécution de la cible

Le fonctionnement d'AFL/QBDI est différent de celui d'AFL dans le sens où l'instrumentation est ajoutée dynamiquement à un binaire déjà compilé. L'idée retenue était de simuler le comportement qu'aurait eu AFL et l'instrumentation statique ajoutée lors de la compilation. Pour y arriver, des *callbacks* sont placés à chaque basic-block. Ils permettent de simuler l'action de l'instrumentation statique d'AFL, à savoir principalement reporter l'état de la cible, et mettre à jour la cartographie du programme via la mémoire partagée (`shared_mem`) avec le fuzzer.

### 10.4.2 Optimisations

À ses débuts, AFL était un fuzzer introduisant des concepts innovants et créatifs. Cependant, l'activité sur ce projet a rapidement décrû depuis ces dernières années, et ce, malgré les avancées non négligeables dans le domaine du fuzzing. Cette absence d'optimisations et d'introduction de nouvelles fonctionnalités empêchent bien souvent le fuzzer d'obtenir des résultats concrets. Afin d'améliorer les campagnes de fuzzing, AFL/QBDI implémente de nombreuses optimisations permettant bien souvent d'atteindre des bugs qu'une version simple d'AFL n'aurait pu découvrir.

**API Fuzzing** La première optimisation proposée par AFL/QBDI et la possibilité de fuzzzer un programme à la manière d'une bibliothèque. En effet, parfois une seule fonction

est intéressante, mais elle nécessite un temps d'exécution important avant de l'atteindre. AFL/QBDI permet de fuzzer directement la fonction souhaitée, et ainsi obtenir des performances parfois bien supérieures.

**Fork-server et cache** A la manière d'AFL, AFL/QBDI propose un mécanisme de **fork-server** adapté au fuzzing d'une cible déjà compilée. Grâce à la présence du **forkserver**, il est possible d'utiliser le mécanisme de cache de QBDI et d'obtenir ainsi des performances étonnantes, malgré l'utilisation d'un moteur d'instrumentation dynamique. En effet, à chaque nouveau basic-block, QBDI désassemble, patche et instrumente le code, avant de réassembler un nouveau basic-block. Ce processus est particulièrement coûteux, et s'il était répété à chaque nouveau processus fuzzé, les performances d'AFL/QBDI en pâtiraient. Ainsi, chaque nouveau basic-block découvert est désassemblé, patché et instrumenté, puis transmis au **fork-server** qui l'ajoutera à son cache. Grâce à cela, toutes les exécutions futures disposeront de ces basic-block en cache.

**Persistance** AFL/QBDI a réimplémenté le mécanisme de persistance proposé par AFL mais l'a adapté au fuzzing d'exécutables déjà compilés. Ce mode est particulièrement efficace lors du fuzzing d'API. En effet, après avoir fait une première exécution de la fonction à fuzzer, au lieu de se servir du forkserver pour créer un nouveau processus, AFL/QBDI conserve le processus courant et réinitialise le contexte d'exécution, afin de lancer l'opération de fuzzing plusieurs fois sans avoir besoin de créer un nouveau processus.

#### Note

La persistance peut avoir un impact, par exemple lorsque le code fuzzé modifie des variables globales.

**Cassage de comparaisons** Une des optimisations proposées par AFL/QBDI est le casseage de comparaisons (voir section ). En effet l'absence d'une telle optimisation dans AFL est une des raisons pour laquelle l'outil n'obtient pas de bons résultats sur un bon nombre de campagnes de fuzzing. Grâce à l'utilisation de QBDI, il est possible d'instrumenter des instructions avec des mnémoniques spécifiques tels que CMP et TEST. On retrouve souvent ces instructions lors d'une comparaison du type `if (x == CONSTANT_VALUE)` qui se retrouvent compilées de la forme `CMP rax, CONSTANT_VALUE`.

Dans le cas d'une telle comparaison (où la variable `x` est contenue dans `rax`), il est particulièrement difficile pour un fuzzer de valider cette condition. En effet, la probabilité de "deviner" la valeur `CONSTANT_VALUE` est extrêmement faible :  $\frac{1}{2^{64}} = 5.4 \times 10^{-20}$ . AFL/QBDI utilise l'information contenue dans l'opérande de l'instruction (`CONSTANT_VALUE`) pour guider le fuzzer vers la génération d'une entrée permettant d'atteindre cette valeur et ainsi casser la comparaison. Plus précisément, AFL/QBDI casse la comparaison de 16, 32 ou 64 bits en de multiples comparaisons de 8 bits. Lorsque l'un des octets concorde avec

la valeur attendue, AFL/QBDI simule alors une progression dans la couverture de code, remontée à AFL.

Note

Cette optimisation est la seule nécessitant une modification du code source d'AFL.

**Bibliothèque d'interposition.** AFL/QBDI dispose d'une bibliothèque d'interposition. Celle-ci est chargée à l'aide de la variable d'environnement `LD_PRELOAD` sous Linux et `DYLD_INSERT_LIBRARIES` sous macOS. Cette bibliothèque tierce permet d'intercepter les appels à des fonctions choisies par l'utilisateur. La modularité de l'outil offre de nombreuses possibilités, et peuvent être appliquées au cas par cas, en fonction de la campagne de fuzzing. Deux exemples génériques ont été développés et sont décrits dans les sections 10.4.2 et 10.4.2.

Note

Le moteur d'instrumentation (QBDI) résidant dans le même espace mémoire que la cible et que la bibliothèque d'interposition, il est important que l'interposition ne s'effectue que lorsque le code de la cible est exécuté, et pas lors des phases où QBDI prend la main. La bibliothèque d'interposition proposée par AFL/QBDI prend en compte cette problématique.

**Cassage de comparaisons de chaînes.** Un des usages de la bibliothèque d'interposition présentée dans la section 10.4.2 est le cassage de comparaisons de chaînes de caractères. En effet, il peut arriver que la cible du fuzzing présente des comparaisons de chaînes de caractères et cela peut s'avérer bloquant dans la progression du fuzzer. En se servant de la bibliothèque d'interposition, une interception des appels aux fonctions usuelles permettant de comparer des chaînes de caractères est effectuée, par exemple, `strcmp`, `memcmp` et `strstr`. A la manière du cassage de comparaisons (voir section 10.4.2), le fuzzer recevra un retour lorsqu'un des caractères de la chaîne a été correctement "deviné". Ainsi, en générant une progression du fuzzer à chaque nouveau caractère atteint, le fuzzer parvient efficacement à atteindre la chaîne de caractère attendue.

**Détection des dépassements sur le tas.** Une autre utilisation de la bibliothèque d'interposition est la détection de dépassements sur le tas. Le principe est similaire à la bibliothèque Dislocatord'AFL. L'outil intercepte les fonctions suivantes :

- `malloc`;
- `realloc`;
- `calloc`;
- `free`;

- `malloc_size`;
- `malloc_good_size`;

L’interception de ces appels permet au fuzzer de modifier la méthode d’allocation, et ainsi détecter les dépassemens de tampons sur le tas.

**Instructions conditionnelles.** Lorsqu’une comparaison intervient dans un programme, le compilateur a bien souvent recours aux instructions assembleurs `CMP` et `TEST`. Suite à cela, se trouve une instruction effectuant le saut suivi de deux basic-block, un dans le cas où la comparaison est valide et un dans le cas où cela ne l’est pas. La présence de ces instructions et les sauts vers des basic-block différents permettent au fuzzer d’obtenir un retour sur la couverture de code, en vérifiant quel chemin a été emprunté. Cependant, les conditions ne sont pas toujours présentées sous cette forme. En effet, on retrouve parfois des instructions dites « conditionnelles ». Elles agissent différemment en fonction de l’état du contexte mémoire. Ces instructions sont problématiques pour les fuzzers, car elles ne génèrent pas de basic-block, le fuzzer n’a donc pas de retour sur le résultat du test conditionnel de l’instruction. Ces instructions conditionnelles (telles que `CMOVcc` et `SETcc`) sont bien souvent bloquantes dans la progression du fuzzer. Afin de résoudre ce problème, AFL/QBDI instrumente spécifiquement les instructions conditionnelles pour générer une progression si la condition est remplie.

#### Note

L’instrumentation des instructions conditionnelles est possible grâce à l’utilisation de QBDI et de sa capacité à instrumenter des instructions en se basant sur leur mnémomique. Lorsqu’est rencontré les instructions `CMOVcc` et `SETcc`, un basic-block « virtuel » est généré, suggérant au fuzzer qu’un nouveau chemin a été découvert.

### 10.4.3 Conclusion

L’intérêt principal d’AFL/QBDI réside dans sa capacité à fuzzer des binaires en source close grâce à QBDI et à l’instrumentation dynamique de binaires. Les modifications apportées grâce à la présence de QBDI apportent des résultats particulièrement intéressants. En effet, l’instrumentation statique proposée actuellement par AFL ne saurait être suffisante pour réaliser les optimisations décrites plus haut. Actuellement le moteur d’instrumentation dynamique ne supporte qu’en partie les architectures ARM. L’implémentation d’une nouvelle architecture reste cependant relativement simple à développer pour QBDI.

## 10.5 Fuzzing #4 : PULSAR

PULSAR est un outil de fuzzing publié en 2015 lors de la conférence *SecureComm*. Le papier de recherche [132] ainsi que l'outil associé [133] ont été développés par une équipe composée de Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, et Konrad Rieck. Il est sous licence *BSD 3* et l'activité du projet est très faible. Il n'y a pas de numéro de version. Ainsi, le code analysé correspond à celui du commit 18b35db5900785d53c1f231fe9110c76e3b7f129. PULSAR est un fuzzer de protocole réseau capable de détecter, comprendre et apprendre automatiquement ce protocole. L'outil prend une approche différente de celles des fuzzers présentés précédemment.

Ce fuzzer est spécifiquement orienté au fuzzing de protocoles réseau. Les tests réalisés par les auteurs semblent indiquer sa capacité à découvrir des vulnérabilités dans différents outils aux protocoles complexes (CVE-2011-4601 dans Pidgin et CVE-2014-4643 dans Core FTP client par exemple). En raison des contraintes de fuzzing réseau prescrites par le projet [PASTIS](#), il était donc nécessaire d'étudier PULSAR.

### 10.5.1 Description de l'outil

PULSAR fonctionne en deux phases. La première consiste à inférer le protocole cible, la seconde à effectuer un fuzzing en *black-box*. La majorité de l'intelligence du fuzzer réside donc principalement dans la première phase. Les étapes majeures de cette phase sont décrites dans les sections suivantes, Inférence du modèle (cf. 10.5.1), Génération des cas de tests (cf. 10.5.1) et Couverture du modèle (cf. 10.5.1).

#### Inférence du modèle

L'étape de l'inférence du modèle est elle même divisée en 5 phases :

1. Acquisition des données ;
2. Groupement (clustering) des messages ;
3. Reconstruction du protocole de la machine à états ;
4. Extraction du format des messages ;
5. Inférence du flux de données.

Le but ultime est de générer un modèle de Markov représentant une machine à états du protocole, accompagné d'un modèle permettant d'identifier le format des messages et un jeu de règles décrivant les communications entre les extrémités.

**Acquisition des données.** L'acquisition des données se fait en capturant le trafic réseau transmis entre les extrémités de la communication, en combinant le couple IP de destination et port. Ces captures peuvent par exemple, être réalisées à l'aide de `wireshark` ou de `tcpdump`. Ces traces réseau composées de messages entre les interlocuteurs sont prétraités par PULSAR, lui permettant d'isoler les informations importantes (identifiants de session par exemple) pour les utiliser dans les étapes suivantes.

**Regroupement (clustering) des messages.** Chaque message est modélisé comme une séquence d'octets inscrits dans un espace vectoriel en vue de leur clustering. Il existe deux types de clustering, dépendant du type du protocole. Le premier se concentre sur les protocoles en format texte tandis que le second vise les protocoles binaires. Dans le cas des protocoles disposant d'informations au format texte ([XML](#) par exemple), chaque jeton du message est associé à une dimension du vecteur de fonctionnalités. Dans le cas des protocoles purement binaires, l'approche est similaire, mais se base sur des [n-grams](#) qui seront utilisés à la place des jetons. Une étape de réduction des vecteurs intervient alors, avant d'établir la distance entre les différents messages vectorisés. Ces distances permettront d'établir un lien entre les messages présentant une structure similaire.

**Reconstruction du protocole de la machine à états.** PULSAR recrée une approximation de la machine à états du protocole étudié, en se basant sur les traces réseau capturées. Cette machine à états est représentée par une chaîne de Markov.

**Extraction du format des messages.** Cette étape assigne à chaque message un état dans la chaîne de Markov. Pour chacun de ces états, un unique groupe de messages comportant le même nombre de jetons est généré. Si tous les messages au sein d'un même groupe se situent à une position spécifique, ce jeton est interprété comme constant, sinon il est considéré comme un champ du message. Cela permet d'obtenir un modèle pour ces messages et l'état associé dans la chaîne de Markov.

**Inférence du flux de données.** L'inférence du flux de données permet de générer un ensemble de règles décrivant les informations contenues dans les différents messages d'une session. PULSAR se sert des messages précédemment analysés pour remplir les différents champs des messages suivants. Pour tout champ  $f$  de ces modèles, PULSAR vérifie l'ensemble de règles générées afin de remplir  $f$  avec des données contenues dans les champs des messages précédents. Si aucune règle ne convient, le jeton est enregistré en tant que nouvelle donnée, et une règle appropriée est alors créée. Il existe cinq règles prédéfinies permettant de remplir les nouveaux champs :

- Copie : Copie l'intégralité du contenu d'un champ dans un autre ;
- Séquence : Copie un champ numérique et l'incrémenter ;
- Addition : Préfixe ou suffixe des données au champ ;
- Partition : Extrait les données d'un champ délimité par un caractère ;
- Données : Utilise les champs précédemment rencontrés pour remplir le champ.

L'étape d'inférence du modèle de PULSAR est la phase essentielle qui en fait un fuzzer à part. En effet, disposer d'un modèle décrivant les communications entre les différents interlocuteurs est un critère déterminant pour le fuzzing de protocoles réseau. La connaissance de la machine à état du protocole permet à PULSAR d'être plus efficace et d'atteindre des chemins plus profonds dans le code.

## Génération des cas de tests

La phase de génération des cas de tests est similaire à l'état de l'art des fuzzers standards. La principale différence réside dans la capacité à générer des messages disposant d'une sémantique correcte. La génération des cas de tests se base sur différentes primitives de fuzzing telles que :

- Séquence UTF-8 invalide ;
- Dépassement de chaîne de caractères ;
- Chaîne de caractères aléatoire (avec ou sans caractères alphanumériques).

La modularité de PULSAR permet d'ajouter facilement de nouvelles primitives. Ces primitives sont utilisées en conjonction du flux de données afin de créer des messages malformés. Le système fonctionne de la manière suivante. Lorsqu'un message est reçu, PULSAR fait appel au modèle afin d'obtenir l'état associé. On se sert de la chaîne de Markov pour générer une transition valide. Le jeu de règles de PULSAR est alors utilisé pour cette transition afin de construire le message suivant. La distance de Levenshtein [232] sur les chaînes de caractères est enfin utilisée pour mesurer les similarités entre les messages reçus et les modèles disponibles à ce stade.

### Note

PULSAR ne dispose d'aucune optimisation à cette étape, contrairement aux fuzzers précédemment étudiés. De plus, le fuzzer fonctionne comme l'un des interlocuteurs, l'application cible nécessite donc d'être lancée manuellement et indépendamment du fuzzer. Cela nécessite aussi une gestion manuelle des éventuels crashes.

## Couverture du modèle

Afin d'atteindre une couverture de code élevée, PULSAR se sert d'un algorithme de sous-graphes pour sélectionner la prochaine réponse de manière efficace. L'algorithme fonctionne comme suit. Dans un premier temps, un masque est appliqué à chaque modèle. Celui-ci consiste en un tableau binaire de taille égale au nombre de champs dans le modèle. Ce tableau permet de déterminer quels champs devront être fuzzés la prochaine fois que ce modèle sera sélectionné pour construire un message. Ensuite, un sous-graphe est défini depuis un état racine, incluant les états atteignables après un nombre de transitions fixé. L'algorithme assigne alors au sous-graphe un poids de fuzzing égal à la somme des poids des différents états. Le poids d'un état est défini par la somme des masques de ses modèles. A chaque fois qu'un message est reçu et identifié, l'état ayant le poids de sous-graphe le plus élevé est sélectionné (parmi ceux représentant une transition valide). Le modèle de réponse est sélectionné en fonction de cet état et de sa probabilité d'occurrence dans les données d'entraînement. Enfin, la communication continue jusqu'à ce qu'un état fuzzable soit atteint. Lorsqu'un modèle est sélectionné pour l'étape de fuzzing, son compteur de masque est décrémenté.

Modifier les masques de fuzzing modifie les champs du modèle qui seront fuzzés la prochaine fois qu'il sera sélectionné. De plus, cela diminue aussi le poids de fuzzing de ses états et des sous-graphes de ses précédents états. Cela entraîne PULSAR à sélectionner en priorité les modèles proposant les meilleures opportunités de fuzzing. Au fur et à mesure que les masques de fuzzing des modèles diminuent, le poids de leurs sous-graphes diminue lui aussi, permettant à d'autres chemins moins prioritaires d'être explorés.

### 10.5.2 Cas d'études

PULSAR présente deux cas d'étude, le premier sur un protocole textuel, le second sur un protocole binaire et propriétaire. Dans les deux cas, PULSAR tente de reproduire des bugs déjà connus. Les tests décrits dans le papier [132] n'ont pu être reproduits, ainsi ce qui suit se base uniquement sur les résultats rapportés par les auteurs.

#### Note

Les auteurs ne fournissent aucune information sur temps nécessaire à la réalisation de l'étude. Ils ne fournissent pas non plus d'indicateurs de comparaison avec d'autres outils.

#### Core FTP Client

Dans un premier temps, le traffic entre l'application cible (le client Core FTP) et le serveur `vsftpd` a été capturé. Très exactement, 987 traces réseau provenant d'interactions standards ont été enregistrées. Cependant, aucune mention n'est faite de la méthode de capture, ni des interactions effectuées, ni du temps nécessaire. PULSAR est parvenu grâce à ces informations à retrouver six vulnérabilités connues de la CVE-2014-4643. Il est aussi parvenu à détecter deux dépassemens de tampon inconnus au préalable.

#### Pidgin ICQ/AIM (protocole OSCAR)

Un scénario plus complexe concerne un exécutable en source fermée reposant sur deux serveurs. Le premier serveur est un serveur d'autorisation, le second un serveur OSCAR. Dans ce cas, 512 traces réseau ont été collectées (les auteurs n'offrent pas plus d'informations à ce propos). PULSAR est parvenu à découvrir la vulnérabilité rapportée dans la CVE-2011-4601

### 10.5.3 Conclusion

PULSAR est un outil intéressant quand il s'agit d'étudier des protocoles réseau. Bien que son stade de développement reste académique et proche de la recherche, il présente

néanmoins des idées et des techniques particulièrement intéressantes pour le projet PASTIS. Cependant, certains points faibles pourraient être améliorés dans le code et dans le papier associé à l'outil. En effet, aucune mention de comparatif aux outils de fuzzing existants n'est faite dans ce papier. Ainsi, il est difficile de se faire une idée de l'efficacité et de la vitesse de PULSAR. De la même manière, aucun commentaire n'est fait sur la collecte des traces réseau (ni même sur comment reproduire les évaluations effectuées par les auteurs). PULSAR est plus intéressant à considérer comme un outil permettant de reconstruire la machine à état du protocole à étudier qu'un fuzzer en tant que tel.

# Évaluation des outils

---

Les résultats des différents outils sont présentés dans les tableaux 11.1, 11.2 et 11.3. Trois des outils sont ici comparés : AFL, AFL/QBDI et Honggfuzz. Ils ont été évalués uniquement sur l'architecture x86\_64. En effet AFL et AFL/QBDI ne proposent pas de support de l'architecture ARM, quant à Honggfuzz, nos tentatives pour le tester sur l'architecture ARM ont été infructueuses dans le temps imparti au test de l'outil. En effet, Honggfuzz utilise la bibliothèque `libunwind` pour la phase de déduplication des crashes. Cette bibliothèque ne semble pas fonctionner correctement sur ARM, et ne parvient pas à extraire la pile appelante. Tous les crashes sont alors considérés comme uniques, rendant le nombre de crashes trop important pour être traités. De plus, leur sauvegarde sur le disque rend instable la campagne de fuzzing.

### Note

L'outil PULSAR ne permettait pas de s'exécuter convenablement sur la suite de tests sans de lourdes modifications à apporter à l'outil.

## 11.1 Bombes logiques

L'objectif des **bombes logiques** est de tester différents types de bugs présentés dans la section 5.2. Les résultats bruts sont les suivants :

- AFL : 46/70 ;
- AFL/QBDI : 45/70 ;
- Honggfuzz : 55/70.

Les résultats d'AFL et AFL/QBDI sont très similaires. Honggfuzz a lui un avantage non négligeable sur les deux autres concurrents, puisqu'il parvient à activer 10 bombes de plus. Certaines bombes logiques n'ont été activées par aucun des fuzzers, pour certaines d'entre elles, l'explication est simple. Sans être exhaustifs, quelques exemples sont détaillés ci-dessous.

Les bombes logiques `cpuid_1` (`CId_31`) et `cpuid_2` (`CId_32`) attendent un retour particulier de l'instruction `cpuid`. Le retour de cette instruction étant spécifique à la machine sur laquelle le fuzzer est exécuté, il est impossible pour un fuzzer d'agir sur le résultat de cette instruction.

Note

Les bombes logiques `CId_31` et `CId_32` permettent d'évaluer la capacité de l'outil à fuzzer des programmes composés d'instructions non déterministes. Cela pourrait poser problème pour un fuzzer basé sur l'émulation du programme cible.

D'autres bombes logiques, telles que `sample_path[1|2|3|4]`, n'ont pas été activées, il reste de difficile de déterminer les raisons pour lesquelles les fuzzers ne sont pas parvenus à les déclencher. Les résultats des bombes logiques semblent favoriser `Honggfuzz`. Il est cependant important de noter que certains cas (détaillés ci-dessous) ont été déclenchés par `AFL` et pas par `Honggfuzz`.

- `loop_sample_1` (`CId_45`);
- `b0f_sample_2` (`CId_56`);
- `stackarray_sm_12_off_by_one` (`CId_63`);
- `use_after_free_2` (`CId_65`).

Les résultats, détaillés par unités, sont présentés dans le tableau ci-après (11.1).

## 11.2 LAVA

Les bombes logiques permettent d'évaluer la capacité des outils à déclencher certains types de bugs, cependant, cela n'est pas représentatif des programmes réels. En effet, le caractère de **test unitaire** des bombes logiques ne prend pas en compte la complexité d'un programme, les conditions qu'il peut présenter ainsi que les chemins nécessaires à explorer avant d'atteindre un bug. Pour mesurer la capacité d'exploration d'un outil, la suite de tests `LAVA` a été utilisée. Les tests ont été réalisés sur les exécutables `base64` et `uniq`. Les résultats bruts de ces tests sont donnés dans le tableau 11.2.

### 11.2.1 AFL et AFL/QBDI

Les résultats d'`AFL` sont médiocres. En effet, l'outil n'est parvenu à déclencher aucun des bugs injectés dans `base64` et `uniq`. Une analyse de la couverture de code sur `base64` a été effectuée, en particulier la fonction `base64_decode_ctx`, où les bugs sont injectés. La capture d'écran 11.1 permet de visualiser cette couverture. (Les basic-block en bleus ont été couverts par les exécutions du fuzzer) Les basic-block en bleus, sont ceux couverts par la campagne de fuzzing d'`AFL`. La couverture apparaît plutôt bonne, et ne permet pas à elle seule d'expliquer la difficulté d'`AFL` à fuzzer cet exécutable.

		Id	Nom	AFL 2.52b		AFL/QBDI		Honggfuzz 1.7	
				x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_1	CE_1	CId_1	array_sample1	✓	-	✓	-	✓	-
		CId_2	array_sample2	✓	-	✓	-	✓	-
		CId_3	array_sample3	✓	-	✓	-	✓	-
		CId_4	array_sample4	✓	-	✓	-	✓	-
		CId_5	array_sample5	✓	-	✓	-	✓	-
		CId_6	array_sample6	✓	-	✓	-	✓	-
		CId_7	file_cp_11	✓	-	✓	-	✓	-
		CId_8	file_posix_cp_11	✓	-	✓	-	✓	-
		CId_9	stack_cp_11	✓	-	✓	-	✓	-
		CId_10	stackarray_sm_11	✓	-	✓	-	✓	-
		CId_11	stackarray_sm_12	✓	-	✓	-	✓	-
		CId_12	stackarray_sm_ln	✗	-	✗	-	✓	-
		CId_13	pointers_sj_11	✓	-	✗	-	✓	-
		CId_14	stackarray_sm_store_11	✓	-	✓	-	✓	-
		CId_15	stackarray_sm_store_12	✓	-	✓	-	✓	-
		CId_16	stackarray_sm_rw_11	✓	-	✓	-	✓	-
UT_2	CE_1	CId_17	atoi_ef_12	✓	-	✓	-	✓	-
		CId_18	pow_ef_12	✓	-	✓	-	✓	-
		CId_19	printint_int_11	✓	-	✓	-	✓	-
		CId_20	rand_ef_12	✓	-	✓	-	✓	-
		CId_21	file_csv	✓	-	✓	-	✓	-
		CId_22	file_posix_csv	✓	-	✓	-	✓	-
		CId_23	pid_csv	✓	-	✓	-	✓	-
		CId_24	malloc_1	✗	-	✗	-	✓	-
		CId_25	memset_1	✗	-	✗	-	✓	-
		CId_26	syscall_alarm	✗	-	✗	-	✗	-
		CId_27	syscalls_open_close	✗	-	✗	-	✓	-
		CId_28	syscalls_read_write	✓	-	✓	-	✓	-
		CId_29	syscall_time	✗	-	✗	-	✗	-
		CId_30	pmccntr	✗	-	✗	-	✗	-
		CId_31	cpuid_1	✗	-	✗	-	✗	-
		CId_32	cpuid_2	✗	-	✗	-	✗	-
UT_2	CE_4	CId_33	string_sample1	✗	-	✗	-	✓	-
		CId_34	string_sample2	✓	-	✓	-	✓	-
		CId_35	string_sample3	✓	-	✓	-	✓	-
		CId_36	string_sample4	✓	-	✓	-	✓	-
		CId_37	propagation_sample3	✗	-	✗	-	✗	-
		CId_38	simple_float1	✓	-	✓	-	✓	-
		CId_39	atof_ef_12	✓	-	✓	-	✓	-
		CId_41	sin_ef_12	✓	-	✓	-	✓	-
		CId_42	ln_ef_12	✓	-	✓	-	✓	-
		CId_43	5n+1_lo_11	✓	-	✓	-	✓	-
		CId_44	collaz_lo_11	✓	-	✓	-	✓	-
		CId_45	loop_sample1	✓	-	✗	-	✗	-
		CId_46	loop_sample2	✓	-	✓	-	✓	-
		CId_47	loop_sample3	✗	-	✗	-	✗	-
UT_3	CE_2	CId_48	df2cf_cp_11	✓	-	✓	-	✓	-
		CId_49	sample_path1	✗	-	✗	-	✗	-
		CId_50	sample_path2	✗	-	✗	-	✗	-
		CId_51	sample_path3	✗	-	✗	-	✗	-
		CId_52	sample_path4	✗	-	✗	-	✗	-
		CId_53	sample_path5	✓	-	✓	-	✓	-
		CId_54	sample_path6	✓	-	✓	-	✓	-
		CId_55	bof_sample1	✓	-	✓	-	✓	-
		CId_56	bof_sample2	✓	-	✓	-	✗	-
		CId_57	stack_bo_11	✓	-	✓	-	✓	-
		CId_58	iof_sample1	✗	-	✗	-	✓	-
		CId_59	iof_sample2	✗	-	✗	-	✓	-
		CId_60	integer_overflow_1	✗	-	✗	-	✓	-
UT_4	CE_3	CId_61	off_by_one_1	✗	-	✗	-	✓	-
		CId_62	off_by_one_2	✓	-	✓	-	✓	-
		CId_62	off_by_one_3	✓	-	✓	-	✓	-
		CId_63	stackarray_sm_12_off_by_one	✓	-	✓	-	✗	-
		CId_64	use_after_free_1	✓	-	✓	-	✓	-
		CId_65	use_after_free_2	✓	-	✓	-	✗	-
		CId_66	format_string_1	✓	-	✓	-	✓	-
		CId_67	format_string_2	✓	-	✓	-	✓	-
		CId_68	format_string_3	✓	-	✓	-	✓	-
		CId_69	read_anywhere	✗	-	✗	-	✗	-
		CId_70	stackoutofbound_sm_12	✗	-	✓	-	✓	-

TABLE 11.1 – Résultats détaillés des benchmarks de fuzzing sur les tests atomiques

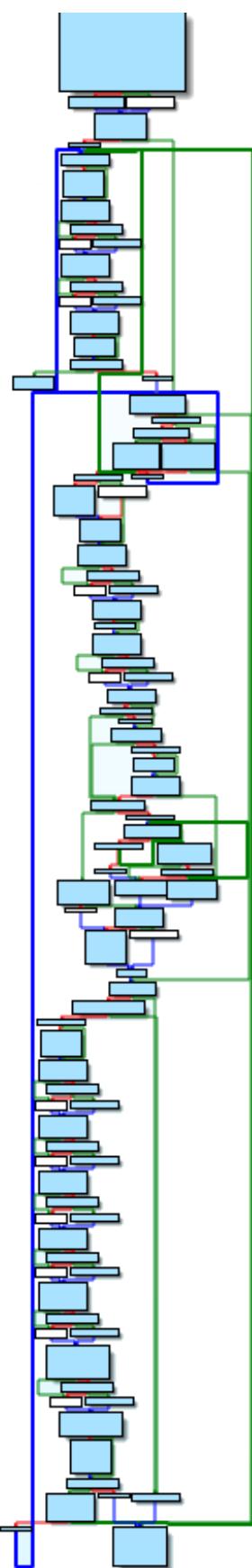
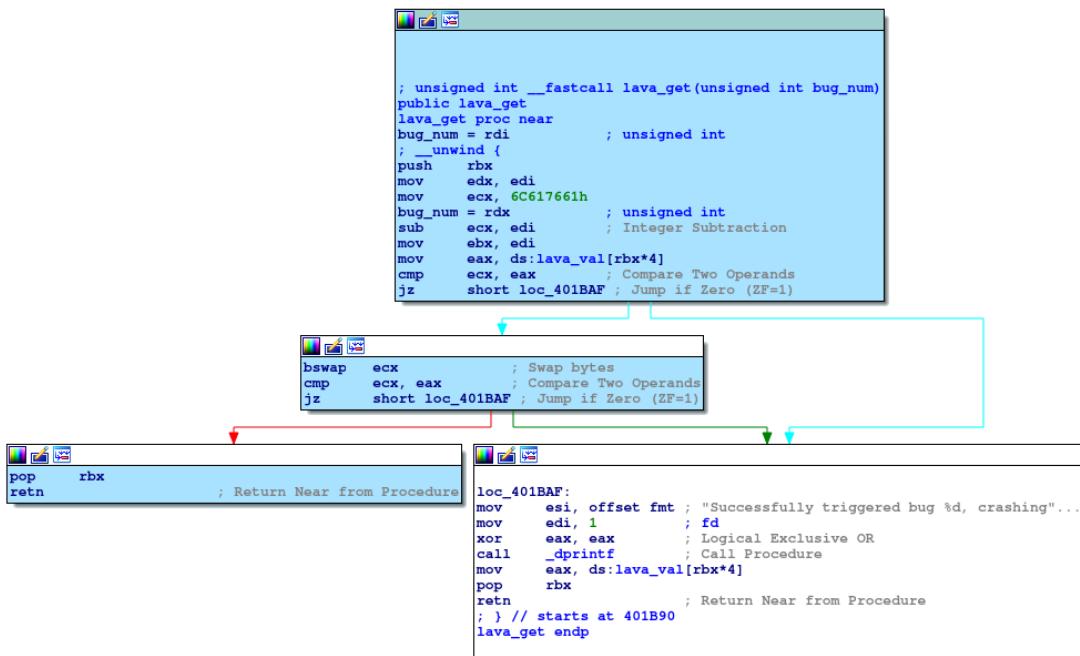


FIGURE 11.1 – LAVA base64- Couverture de code AFL

	CE	Nom	AFL 2.52b		AFL/QBDI		Honggfuzz 1.7	
			x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_5	CE_1 : Nombre de bugs trouvés	base64	0/44	-	18/44	-	24/44	-
		uniq	0/28	-	15/28	-	22/28	-

TABLE 11.2 – Résultats des benchmarks pour UT\_5 : Passage à l'échelle

En s'intéressant davantage au mécanisme d'injection de bugs de LAVA, celui-ci emploie les fonctions `lava_set` et `lava_get`. Elles permettent respectivement de consommer l'entrée utilisateur, et de la comparer à la valeur attendue afin de déclencher le bug. La capture 11.2 est relativement explicite. La couverture de la fonction `lava_get` n'est pas complète, en effet le seul basic-block non couvert (en blanc), est celui responsable du bug. Ce basic-block est conditionné par la résolution d'une contrainte, qui dépend de l'entrée utilisateur.

FIGURE 11.2 – LAVA base64 (`lava_get`) - Couverture de code AFL

La résolution de cette contrainte nécessite de “deviner” un `magic number` sur 32 bits. AFL ne dispose d'aucune indication sur la manière de deviner cet entier, et se contente donc uniquement de bruteforcer l'entier sur tout l'espace de 32 bits. Cette contrainte se pose pour tous les bugs injectés par LAVA, expliquant donc le score nul d'AFL.

## 11.2.2 Honggfuzz

Honggfuzz démontre des performances bien supérieures à AFL. En effet, il semble à priori déclencher plus de la moitié des bombes sur `base64`, et presque toutes sur `uniq`.

Note

AFL/QBDI permet le déclenchement de certains des bugs, car il dispose d'une optimisation permettant le cassage de comparaisons, comme décrit dans la section 10.4.2. Les travaux autour d'AFL/QBDI permettant d'améliorer cette fonctionnalité ont récemment été publiés [116].

Des travaux effectués en parallèle du projet PASTIS ont permis de pousser l'analyse des résultats fournis Honggfuzz. En effet, d'autres cas ont mis en évidence des erreurs dans le mécanisme de tri des crashes de l'outil (celui-ci se base sur libunwind). Afin de parfaire cette analyse, une analyse détaillée a été effectuée, non seulement des entrées ayant généré des crashes, mais aussi de l'intégralité des entrées ayant généré une couverture de code supplémentaire. L'ensemble des entrées du corpus est un surensemble des entrées des crashes. En testant manuellement les entrées contenues dans le corpus, celles-ci devraient déclencher les mêmes `bug_id` que Honggfuzz lors de sa campagne de fuzzing.

Dans le cas d'exemple, l'ensemble des entrées des crashes contient 25 fichiers, mais ne permet de déclencher que 18 bugs ayant des `bug_id` différents<sup>138</sup>.

Un premier constat peut être tiré de ces résultats : l'identification des bugs uniques par Honggfuzz n'est pas en adéquation avec la définition d'un bug unique au sens de LAVA. En effet, certains bugs ont été identifiés comme unique par Honggfuzz, générant 25 fichiers de crashes différents, tandis que LAVA n'en identifie que 18 uniques. Honggfuzz "sur identifie" certains bugs, en émettant plus de "bugs uniques" que LAVA.

La même analyse a été effectuée sur l'ensemble des entrées du corpus contenant 526 fichiers. Ces fichiers permettent de déclencher 48 bugs ayant des `bug_id` différents<sup>139</sup>. Il est important de noter qu'Honggfuzz a déclenché 48 bugs, là où LAVA indique n'avoir inséré que 44 bugs.

Ces résultats corroborent le premier constat, mais cette fois, certains bugs qui auraient dû être identifiés comme uniques (au sens de LAVA), ont été considérés comme déjà rencontrés par Honggfuzz, et n'ont pas été sélectionnés comme bug unique, mais uniquement comme entrée ajoutant de la couverture de code. Honggfuzz a donc ici, "sous-identifié" les bugs uniques.

Le processus de `crash triaging` d'Honggfuzz ne semble pas s'accorder avec la définition d'un "bug unique", tel que l'entend LAVA, générant des résultats moins bons que la réalité. Il est difficile de déterminer si l'une ou l'autre des définitions d'un bug unique est correcte sans une analyse plus détaillée du framework LAVA et du processus du tri de crash de Honggfuzz.

Pour Honggfuzz, un gros différentiel de bugs était trouvé d'une exécution à une

138. (1, 222, 235, 253, 255, 276, 278, 386, 554, 556, 558, 560, 562, 566, 576, 786, 815, 843)

139. (1, 222, 235, 253, 255, 274, 276, 278, 284, 386, 521, 526, 527, 554, 556, 558, 560, 562, 566, 572, 573, 576, 582, 583, 584, 774, 776, 778, 780, 782, 784, 786, 788, 790, 792, 798, 804, 805, 806, 813, 815, 817, 831, 832, 835, 841, 842, 843)

autre. Comme pour les autres fuzzer le résultat est l’union des identifiants de bugs. Pour **Honggfuzz** chaque exécution individuelle exhuma environ moitié moins de bugs que le résultat combiné. Cette observation confirme l’intérêt d’exécuter plusieurs fois un test pour “lisser” l’aspect purement aléatoire des mutations.

## 11.3 Synthèse & Conclusion

Le tableau 11.3 permet d’avoir une vue synthétique de l’ensemble des tests.

	Critères	AFL 2.52b		AFL/QBDI		Honggfuzz 1.7	
		x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_1	CE_1 : Tableau	6/6	-	6/6	-	6/6	-
	CE_2 : Propagation	3/3	-	3/3	-	1/3	-
	CE_3 : Lecture mémoire symbolique	3/4	-	2/4	-	4/4	-
	CE_4 : Ecriture mémoire symbolique	2/2	-	2/2	-	2/2	-
	CE_5 : Lecture/Ecriture mémoire symbolique	1/1	-	1/1	-	1/1	-
UT_2	CE_1 : Fonctions externes	7/9	-	7/9	-	9/9	-
	CE_2 : Appels système	1/4	-	1/4	-	2/4	-
	CE_3 : Instructions non déterministes	0/3	-	0/3	-	0/3	-
	CE_4 : Gestion des strings	3/5	-	3/5	-	4/5	-
	CE_5 : Nombres flottants	4/4	-	4/4	-	4/4	-
UT_3	CE_1 : Boucles	4/5	-	3/5	-	3/5	-
	CE_2 : Chemins	3/7	-	3/7	-	3/7	-
UT_4	CE_1 : débordement de tampon	3/3	-	3/3	-	2/3	-
	CE_2 : débordement d’entiers	0/3	-	0/3	-	3/3	-
	CE_3 : Off-by-one	3/4	-	3/4	-	3/4	-
	CE_4 : Use after free	2/2	-	2/2	-	1/2	-
	CE_5 : Format string	3/3	-	3/3	-	3/3	-
	CE_6 : Accès mémoire invalide	0/2	-	1/2	-	1/2	-
Total bombes logiques :		48/70	-	47/70	-	54/70	-
UT_5	CE_1 : Nombre de bugs trouvés base64	0/44	-	18/44	-	24/44	-
	CE_1 : Nombre de bugs trouvés uniq	0/28	-	15/28	-	22/28	-
Total :		48/142	-	80/142	-	100/142	-

TABLE 11.3 – Résultats synthétiques des benchmarks des outils de Fuzzing

L’étude des différents projets de fuzzing pour l’état de l’art nous permet de nous positionner vis-à-vis de l’outil le plus adapté au projet **PASTIS**. Dans un premier temps, l’outil le moins adapté au projet semble être **PULSAR**. Même si celui-ci émet des idées intéressantes pour un fuzzing réseau, il reste difficile d’imaginer prendre ce projet en main, et y apporter les modifications nécessaires pour **PASTIS**. De plus, l’état actuel de celui-ci ne nous a pas permis d’évaluer son efficacité face à la suite de tests.

Concernant **AFL**, il reste la référence dans les projets de fuzzing publics. Il a de nombreuses fois fait ses preuves, sur sa capacité à trouver des bugs dans divers projets. Cependant, nous ne le recommandons pas pour le projet **PASTIS**. En effet, son manque

de fonctionnalités modernes, l'architecture de son code, ainsi que l'état de maintenance du projet sont un frein au développement serein d'un projet en dépendant. Ses résultats sur la suite de tests étaient partagés : Relativement bons face aux bombes logiques, ils restent médiocres face à LAVA, dû à l'absence de la fonctionnalité permettant de casser les comparaisons.

AFL/QBDI obtient des résultats supérieurs à AFL lors des tests réalisés sur LAVA, notamment grâce à sa fonctionnalité de cassage des comparaisons 10.4.2. Cependant, même si l'ajout d'une DBI permet de nouvelles fonctionnalités, l'avantage majeur que représente AFL/QBDI est sa capacité à fuzzzer des exécutables en source fermée, ce qui n'est pas un critère de décision majeur pour le projet PASTIS.

L'étude et les benchmarks réalisés présentent Honggfuzz comme étant le choix le plus prometteur, puisqu'il a été efficace à la fois contre les bombes logiques, mais aussi contre LAVA. L'architecture de son code, ses fonctionnalités modernes, et l'activité du projet favorisent grandement son choix. Si Honggfuzz reste l'outil de fuzzing le plus recommandé, il n'en reste pas moins quelques zones d'ombres qui n'ont pas pu être analysées en détail lors de cette étude. En effet, bien qu'il soit le seul fuzzzer supportant théoriquement l'architecture ARM, une erreur lors des tests a été bloquante pour tester l'efficacité de cette architecture. De plus, ses capacités de fuzzing réseau, bien qu'existantes, n'ont pas été étudiées en détail pour cet état de l'art. Les commits récents du projet semblent indiquer un intérêt particulier pour le fuzzing réseau, et Honggfuzz a déjà fait ses preuves, en trouvant des bugs sur des projets tels que Apache et Nginx.

Le fuzzzer choisi nécessitera dans tous les cas, une adaptation du code aux besoins du projet. La qualité et la modularité du code d'Honggfuzz faciliteront grandement l'évolution de celui-ci.

# **Cinquième partie**

## **Combinaisons d'analyses**



## Chapitre 12

---

# État de l'art : Combinaisons d'analyses

---

## 12.1 Introduction

Après avoir décrit le fonctionnement des techniques et outils d'analyse concolique, de fuzzing et de slicing, cette partie présente les différentes techniques servant à les combiner. Cette tâche est particulièrement délicate, car elle ajoute aux problématiques de recherche de chaque technique, de nouvelles problématiques liées aux communications entre ces approches. Cet état de l'art présente donc les différentes problématiques rencontrées ainsi que les solutions proposées pour les résoudre et tirer le meilleur profit de chaque analyse. Le CCTP considère les combinaisons d'analyses suivantes :

- statique + concolique, noté `COMB_SC` ;
- concolique + fuzzing, noté `COMB_CF` ;
- statique + concolique + fuzzing, noté `COMB_SCF` ;
- statique + slicing + concolique + fuzzing, noté `COMB_SICEF`.

Cet état de l'art se focalise sur `COMB_CF` dans la mesure où l'analyse statique effectuée dans ce projet se base principalement sur `Klocwork`. Évidemment, toute analyse statique associée à une combinaison sera étudiée.

### Note

Cet état de l'art considère une notion d'exécution concolique élargie dans la mesure où ni `Angora` ni `Eclipser` n'effectuent de l'exécution concolique *stricto sensu*.

Les combinaisons `COMB_CF` revêtent différentes appellations selon le contexte et les personnes qui les ont employées. Pour désigner ce type de combinaison, le terme *Hybrid Concolic Testing* [245] a été introduit par Koushik Sen en 2007. La même année le terme d'*analyse compositionnelle* [7, 135] a été introduit par Patrice Godefroid qui en 2012 a introduit le nom de *whitebox fuzzing* [139]. Plus récemment, le terme d'*analyse compositionnelle* a été réutilisé dans différents travaux [269]. Le terme recherche générati<sup>on</sup>nelle *generational search* a aussi été employé [138, 139]. Différentes idiosyncrasies en termes

de combinaisons se cachent derrière ces différentes dénominations, mais toutes résultent d'une combinaison de techniques impliquant de l'exécution symbolique.

## 12.2 Problématiques

La principale problématique posée par les outils de fuzzing, est la capacité de couvrir du code avec des performances (presque) natives, mais qu'ils ne sont capables que de résoudre des conditions de branchement assez “lâches” (libre) comme  $x > 0$ . Le DSE à l'inverse peut résoudre des conditions très précises du type  $x == 0xCAFEBABE$  représentant une unique valeur sur un espace de recherche de  $2^{32}$ .

Un second problème est la gestion des états symboliques. Un DSE gère généralement de nombreux états correspondant chacun à un chemin du programme. Pour alterner les états il doit utiliser un mécanisme de *snapshot* permettant de sauvegarder et restaurer ces états. Les *snapshots* permettent notamment à deux états lors d'un fork de partager une partie de l'état (avec du *copy-on-write* etc). Or, ce mécanisme se combine assez mal avec le fuzzing car peut-être relativement lent. De plus, le fuzzing ne peut pas tirer profit de ce mécanisme (*par ailleurs coûteux en mémoire*).

## 12.3 Fuzzers Whitebox

Le *fuzzing whitebox* désigne la méthode visant à entrelacer du fuzzing avec une analyse sémantique du programme. Cette analyse sémantique se matérialise soit pour effectuer de la couverture, soit pour effectuer la mutation des entrées.

Pour de la couverture, l'analyse sémantique peut se matérialiser par de la propagation de teinte [63, 71, 131] ou encore du DSE pour résoudre des contraintes de chemin [139]. La problématique de recherche posée par cette combinaison est la difficulté de concilier le passage à l'échelle naturelle du fuzzing avec le DSE beaucoup plus coûteux par essence (noté RQ\_COMB1). Néanmoins, cette combinaison s'avère payante et d'impressionnantes cas de passage à l'échelle ont été démontrés [44]. La différence entre DSE et fuzzing whitebox basé sur le DSE est ténue. En effet, un DSE basé sur une exécution concrète du programme effectue en essence la tâche d'un fuzzer. La seule vraie différence est qu'un fuzzer cherche systématiquement des vulnérabilités alors qu'un DSE a une visée plus large de test logiciel et de couverture [246].

Le fuzzing whitebox pose la problématique de l'alternance entre fuzzing et exécution symbolique. Pour ça Driller [333] propose le DSE sélectif qui déclenche l'exécution symbolique dès que le fuzzer est bloqué selon des critères qu'il définit (cf. 13.2). DigFuzz [394] propose un modèle probabilistique de priorisation de chemins. L'idée est d'échantillonner l'exécution du programme avec un corpus initial. Avec les résultats, il est possible d'établir les chemins peu empruntés et difficiles d'accès. Ceux-ci seront donc envoyés en priorité à

l'exécuteur symbolique plus à même de les résoudre. Pour se faire il combine AFL, `angr` et *MCP<sup>3</sup>* (*Monte-Carlo Based Probabilistic Path Prioritization Model*).

Pour la génération d'entrée, certains fuzzers comme GRT [244], TaintScope [356] ou l'article [63] cherchent, via une analyse sémantique, des informations concernant la structure des entrées pour ensuite effectuer du fuzzing *grey-box* classique. Cela permet notamment de définir des contraintes sur les entrées qui peuvent être utilisées par le fuzzer dans son algorithme de mutation. Le tableau 12.1 référence la plupart des fuzzers whitebox de la littérature. Certains comme Qsym et Driller en tirent leurs caractéristiques car ils sont utilisés conjointement avec AFL.

### 12.3.1 Fuzzing guidé

Le but d'un fuzzer whitebox est principalement de pouvoir guider le fuzzer vers des zones précises du programme ou inversement de lui éviter de perdre du temps, par exemple, sur des calculs de sommes de contrôle et autres hachés.

VUzzer [290] et GRT [244] effectuent tous deux des analyses statiques/dynamiques pour guider la couverture du fuzzer. À la compilation Dowser [160] effectue une analyse statique pour identifier les potentiels bugs de déréférencement situés dans les boucles. Il utilise ensuite une analyse de teinte pour trouver la relation entre les boucles et les entrées du programme. Il couvre ensuite uniquement les instructions teintées avec du DSE pour passer à l'échelle. D'autres approches utilisent aussi de la teinte [113, 188] ou encore TaintScope [356] l'utilisent pour trouver les octets "chauds", dits : "*hot bytes*" de l'entrée qui mènent à certains appels systèmes ou appels de bibliothèques critiques. BuzzFuzz [buzzfuzz] fonctionne selon le même principe. Angora [71] améliore cette analyse en associant à chaque contrainte du chemin l'octet associé pour ensuite effectuer une recherche sur le prédicat complet via une descente de gradient (cf. Section 13.3).

Ces approches sont aussi utilisées pour éviter les sommes de contrôle *checksums* très limitantes pour un fuzzer [54] (*et de surcroît très présentes dans les protocoles réseau*). TaintScope [356] adresse ce problème avec une analyse de teinte puis *patch* le programme. Lors de la génération de l'entrée, il calcule puis restaure la valeur de checksum dans celle-ci. Enfin T-Fuzz [276] généralise l'idée avec la notion de *Non-Critical Checks* (noté NCC) qui sont tous les branchements pouvant être sautés sans casser la logique du programme.

## 12.4 Cyber-Reasoning-System

Un CRS est un système *complètement autonome* faisant intervenir différentes analyses combinées pour faire de la recherche de vulnérabilités. Cependant, ce qui le différencie d'un fuzzer *whitebox* est l'ajout des fonctionnalités suivantes :

- capacité de "patcher" ses propres programmes avec d'éventuelles vulnérabilités

	Général	Pré-trait.	Géné. d'entrées	Exécution de la cible
	Open-source	Code source requis	Instrumentation	Mise à jour
	Statique	Dynamique	Ordonnancement	Réduction taille
	Modèle	Type	Contraintes(SMT)	
Angora [71]	✓			
BitFuzz [54]		✓		
BuzzFuzz [131]		✓		
CAB-Fuzz [207]			✓	
Chopper [345]	✓	✓	✓	
DigFuzz [394]		✓	✓	
Dowser [160]				
Driller [333]	✓		✓	
Eclipser [78]	✓		✓†	
Fuzzgrind [61]	✓		✓	
GRT [244]		✓	✓	
MoWF [282]			✓	
MutaGen [201]			✓	
Narada [303]	✓	✓	✓	
Qsym [380]	✓		✓	
redqueen [13]	✓		✓*	
SAGE [139]			✓	
TaintScope [356]			✓	
T-Fuzz [276]	✓	✓	✓	✓
VUzzer [290]	✓	✓	✓	✓

\* : se veut beaucoup plus légère qu'une teinte classique, † : sans résolutions de formules

TABLE 12.1 – Comparaison des caractéristiques des outils de Fuzzing Whitebox

- capacité d'attaquer d'autres CRS avec des vulnérabilités trouvées (*ce qui implique la capacité de générer des exploits*)
- capacité de se protéger et de réagir à une attaque

Cette dénomination a été introduite lors du challenge CGC et donc seuls les outils y ayant participé utilisent ce terme. Au-delà de leurs capacités d'attaque et de défense, ils se composent nécessairement autour de combinaisons d'analyses comprenant pour la majorité du fuzzing et de l'exécution symbolique. Le tableau 12.2 référence les différentes équipes et CRS ayant concouru, les résultats obtenus aux qualifications ainsi qu'à la finale ayant eu lieu le 4 Août 2016 à Las Vegas<sup>140</sup>.

CRS	Équipe	Affiliation	Financé DARPA	open-source	DSE	Fuzzing	Score Qualif.	Score Finale
 Mayhem	ForAllSecure	ForAllSecure	✓	✗	Mayhem	n/c	1 <sup>er</sup>	1 <sup>er</sup>
 Galactica	CodeJitsu	UC.Berkley	✓	✗	S2E	AFLfast	3 <sup>ème</sup>	5 <sup>ème</sup>
 Mechaphish	Shellphish	UC. Santa Barbara	✗	✓✓	angr	Driller	7 <sup>ème</sup>	3 <sup>ème</sup>
 Rubeus	Deep Red	Raytheon	✗	✗	n/c	n/c	5 <sup>ème</sup>	4 <sup>ème</sup>
 Jima	CSDS	CSDS, Moscow	✗	✗	n/c	n/c	2 <sup>ème</sup>	6 <sup>ème</sup>
 Crspy	Disekt	Athène, Grèce	✗	✗	S2E	n/c	6 <sup>ème</sup>	7 <sup>ème</sup>
 Xandra	TECHx	GrammaTech Inc.	✓	~	Grace	AFL *	4 <sup>ème</sup>	2 <sup>ème</sup>
cyberdyne	TrailofBits	TrailofBits	✓	✓	KLEE §	GRR	9 <sup>ème</sup>	-
n/c	Cyberjujitsu	n/c	n/c	n/c	n/c	n/c	11 <sup>ème</sup>	-
n/c	DeFENCE	n/c	n/c	n/c	n/c	n/c	12 <sup>ème</sup>	-
n/c	DESCARTES	n/c	n/c	n/c	n/c	n/c	13 <sup>ème</sup>	-
n/c	FuzzBOMB	n/c	n/c	n/c	n/c	n/c	10 <sup>ème</sup>	-
n/c	Lekkertech	n/c	n/c	n/c	n/c	n/c	8 <sup>ème</sup>	-

§ : mais aussi PySymEmu, \* : version modifiée de AFL

TABLE 12.2 – Équipes et résultats des CRS au Cyber Grand Challenge

La plupart des éléments clés du fonctionnement de **Mayhem** ont été fournis en Section.7.2 et une étude approfondie de **cyberdyne** [142] est donnée en Section.13.5 de cette partie. Pour le reste des CRS, peu de documentation existe [328] et quelques articles de blog de GrammaTech viennent fournir des informations sur les fonctions des différents CRS<sup>141</sup>. Eux utilisent par exemple une version modifiée d'AFL pour le fuzzing personnalisé afin d'obtenir une meilleure couverture et de s'interfacer avec leur exécuteur symbolique *Grace*. Pour la recherche de vulnérabilité et la génération d'exploit, tous les compétiteurs de la finale ont employé un mélange d'exécution symbolique et de fuzzing (*ce qui n'était*

140. <https://www.youtube.com/watch?v=n0kn4mDXY6I>

141. <https://blogs.grammatech.com/the-cyber-grand-challenge>

pas un prérequis de la compétition). Cela confirme l'utilité de telles combinaisons pour de la recherche de vulnérabilités.

## 12.5 Analyse compositionnelle

Cette notion introduite par P. Godefroid [7, 135] vise à traiter chaque fonction d'un programme indépendamment via une première analyse. Puis dans un second temps, le programme est traité dans sa globalité. Dans le cas de [7, 135] ; des pré/post-conditions sont déduites de chaque fonction puis celles-ci sont réutilisées dans la phase de test/fuzzing du programme en entier. Plus récemment, l'outil **MACKE**<sup>142</sup> [267, 269] et son successeur **wildfire** [268] ont proposé une analyse statique (modulaire) de chaque fonction combinée ensuite à une exécution concolique, modifiée pour effectuer une exploration interprocédurale dirigée. Plus précisément, les vulnérabilités sont identifiées statiquement (sur l'IR de LLVM) puis le SE cherche un chemin faisable dans le CFG interprocedural qui permettrait d'activer cette vulnérabilité via une stratégie d'exploration s'appuyant sur du SDSE [243] (*shortest-distance symbolic execution*) (cf. Section 6.7.2).

## 12.6 Etat de l'art : Conclusion intermédiaire

Différentes approches et outils de combinaison ont été évoqués dans cet état de l'art. D'autres comme **Symbiotic** [323] ont été évoquées dans la partie Slicing (cf.15). Cet état de l'art, loin d'être exhaustif dépeint néanmoins les différentes questions de recherches soulevées par les combinaisons de type COMB\_CF. Le tableau 12.3 synthétise ses différentes problématiques.

Id	Description
RQ_COMB1	Comment concilier le passage à l'échelle du fuzzing au DSE réputé moins apte à passer à l'échelle ?
RQ_COMB2	Quels critères et métriques indiquent qu'il est préférable d'interchanger le mode d'analyse fuzzing vers DSE (et vice-versa) ?
RQ_COMB3	À partir d'une entrée générée par un fuzzer comment « resymboliser » certains octets pour non pas, suivre le même chemin, mais en découvrir de nouveaux ( <i>semi-symbolic inputs</i> ) ?

TABLE 12.3 – Récapitulatif des questions de recherche posées par les combinaisons d'analyses

La question de recherche RQ\_COMB2 trouve consensus à base de compteur “*hit-count*” sur les conditions qu'essayent de traverser le fuzzer. A l'inverse, RQ\_COMB1 reste problématique. Dans l'état de l'art, les trois principales solutions au passage à l'échelle sont :

142. <https://github.com/tum-i22/macke>

- la fusion d'états *path-merging* [16] (cf. 6.7.5)
- une exécution symbolique très sélective [333] pouvant être assistée par une analyse statique approfondie (*comme pour Firmalice* [319])
- une exécution symbolique sous-contrainte [117, 288] (relaxée)

C'est pour cela que certaines approches favorisent l'analyse de teinte seule comme **Angora** [71] ou bien des contraintes relaxées (*et donc potentiellement incorrect*) comme pour **Qsym** [380], voir même du DSE sans résolutions de contraintes pour **Eclipser** [78]. Ces trois outils sélectionnés pour l'étude sont décrits en détail dans le chapitre suivant (13). Les dernières approches tel que **SLF** [377] et **redqueen** [13] aspirent même à se passer complètement d'analyse de teinte ou de DSE pour déterminer avec des analyses aussi légères que possible les relations entre les entrées et les états du programme les utilisant.



## Chapitre 13

---

# Analyse détaillée des outils

---

## 13.1 Outils de combinaisons sélectionnés

L'état de l'art effectué précédemment a permis d'isoler plusieurs outils dignes d'intérêt dans le contexte du projet. Le tableau 13.1 présente les caractéristiques des outils sélectionnés à partir des critères d'évaluations des outils de fuzzing (CTO\_F) et des outils de DSE (CTO\_C). Pour des raisons de clarté, certains critères ont été filtrés pour ne conserver que les plus importants.

Le choix s'est naturellement tourné vers Driller [333], dont la publication en 2016 a eu un retentissement certain, en particulier de par sa participation au CGC. cyberdyne a aussi été sélectionné car c'est le seul autre compétiteur du CGC à avoir ouvert les sources des différents composants de son CRS. Par ailleurs, leurs résultats au CGC sont un gage de fiabilité pour la capacité à passer à l'échelle. Malheureusement, aucun des deux outils n'a pu être testé, car tous deux fonctionnent par défaut sur le système Linux DECREE et nécessitent un effort pour les adapter à Linux qu'il n'était pas possible de faire dans le temps imparti.

En outre, Qsym , Angora et Eclipser ont été sélectionnés, car ils marquent un renouveau dans les approches de combinaisons qui sont faites aujourd'hui. Celles-ci supplantent la sacro-sainte correction *soundness* du DSE par des approches relaxées *unsound* permettant un passage à l'échelle jusqu'alors inatteignable. Ce sont, par ailleurs, les publications les plus récentes (2018 et 2019) et les outils sont tous trois open-source. Eclipser n'est même pas encore publié, puisqu'il devrait l'être officiellement le 31 Mai 2019 à ICSE à Montréal <sup>143</sup>.

---

143. <https://2019.icse-conferences.org/event/icse-2019-technical-papers-grey-box-\concolic-testing-on-binary-code>

		#1 Driller	#2 Angora	#3 Qsym	#4 cyberdyne	#5 Eclipsor
CTO	CTO_1 : Langage	Python	Rust	C++,Py	Python	F*
	x86	✓	✓	✓	✓	✓
	x86-64	✓	✓	✓	✓	✓
	ARMv7	~	✗	✗	✗	✗
	ARMv8	~	✗	✗	✗	✗
	CTO_3 : Open-source	✓	✓	✓	✓	✓
	CTO_4 : Licence	BSD-2	Apache 2	n/c	Apache 2	MIT
CTO_F	CTO_5 : Documentation	✓	✗	✗	✗	~
	CTO_6 : Activité	✗	~	✗	✗	~
	CTO_7 : Jeu de tests	✓	✓	✓	✓	✓
	CTO_F1 : Granularité	○	○	○	○	○
	CTO_F2 : Code requis	✗	✓	✗	✗	✗
	CTO_F3 : Instrumentation	statique	✗	✗	✗	✗
		dynamique	✓	✗	✓	✓
CTO_C	CTO_F5 : Modèle	grammaire	✗	✗	✗	✗
		inféré	✗	✗	✗	✗
	CTO_F6 : Mutation	aléatoire	✗	✗	✗	✗
		intelligente	✓	✓	✓	✓
	CTO_F10 : Sanitizer	spatial	✓	✓	✓	✓
		temporel	✓	✓	✓	✓
	Exéc.					
Formule	CTO_C1 : Base	source	✗	✓	✗	✗
		binaire	✓	✗	✓	✓
	CTO_C2 : Type	symbolique	✓	✗	✗	n/a
		concolique	✗	✓	✓	n/a
	CTO_C6 : Représentation intermédiaire	VEX	n/c	∅	LLVM	n/c
	CTO_C11 : Lecture symbolique	✓	n/c	n/c	✓	n/c
		✓	n/c	n/c	✓	n/c
	CTO_C13 : Procédure décision	✓	n/c	n/c	✓	n/c
Formule	CTO_C16 : Optimisations	slicing	✗	✗	✗	✗
		tainting	✓	✓✓	✓	✗
	CTO_C17 : Stratégie couverture	✓	✓	✓	✓✓	✓
	CTO_C21 : Solveurs supportés	Z3,CVC4	∅	Z3	*	∅
		bit-vecteurs	✓	✓	✓	∅
		tableaux	✓	∅	n/c	∅

TABLE 13.1 – Comparaison des outils de combinaisons sélectionnés

## 13.2 Combinaison #1 : Driller

**Driller** est un fuzzer hybride combinant une exécution concolique (basée sur `angr` [320]) et du fuzzing avec `AFL` (en mode `QEMU`). Le premier, utilisé de manière “sélective” sert essentiellement à générer de nouvelles entrées difficiles pour `AFL`. Cet outil a été publié dans le contexte de la compétition du CGC et du `CRS Mechanical Phish` [321]. Il a donc été conçu pour fonctionner sur le Linux créé par la `DARPA` (DECREE). Il a été développé par l'université de Santa Barbara qui l'a publié sur Github<sup>144</sup> en version 1.0 le 21 août 2018. Cette étude se base sur la publication fondatrice suivante :

“Driller : Augmenting Fuzzing Through Selective Symbolic Execution” par Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna publié au 23rd Network and Distributed System Security Symposium, NDSS 2016

### 13.2.1 Introduction

En raison du faible passage à l'échelle des outils de DSE, les auteurs défendent l'idée que les bugs trouvés sont peu profonds et superficiels.

Leur approche est basée sur l'observation qu'il y a principalement deux catégories d'entrées : les entrées générales *general-input* ayant un large éventail de valeurs valides (e.g le nom d'utilisateur ou un mot de passe) et les entrées spécifiques *specific-input* ayant un nombre très réduit de valeurs (e.g hash etc). Ils introduisent alors la notion de **compartiment** dans un programme où chaque compartiment traite un type d'entrées différentes et où les compartiments spécifiques se trouvent dans les compartiments généraux.

De cette manière, selon les auteurs, le fuzzing serait compétent à résoudre les entrées générales, mais échouerait à pénétrer les compartiments spécifiques. Ainsi, l'exécution symbolique sélective permettrait de pénétrer ces compartiments alors que l'explosion combinatoire de chemin l'empêcherait de couvrir les compartiments généraux. L'idée est donc de combiner les approches pour que le fuzzer explore les compartiments initiaux et, lorsqu'il est bloqué, rende la main à l'exécuteur symbolique pour le guider vers de nouveaux compartiments et ainsi de suite. Suivant cette intuition, **Driller** permet de guider la mutation génétique d'`AFL` vers des bugs *profonds* dans le binaire.

### 13.2.2 Travaux connexes

**Driller** se défend d'être du fuzzing hybride dans la mesure où cette méthode ne distingue pas les “checks” simples et difficiles (généraux et spécifiques) et ne permet pas de progresser profondément dans le programme en découvrant les compartiments. De

---

144. <https://github.com/shellphish/driller>

plus, par son approche, **Driller** limite au maximum le fork d'états alors que ce n'est pas le cas du fuzzing hybride (simple). Il se différencie aussi du whitebox fuzzing, dans la mesure où celui-ci va couvrir symboliquement les deux branches d'une condition si elles sont faisables alors que **Driller** n'effectue du DSE que pour passer d'un compartiment à un autre. Ainsi aucune énergie n'est perdue à faire de la couverture symbolique au sein d'un compartiment. Ces différences sur le nommage de la technique n'ont d'autre intérêt que de se distinguer de l'état de l'art dans le processus d'acceptation d'un article.

### 13.2.3 Fonctionnement

Conceptuellement une application est divisée en compartiments. Le flot d'exécution alterne entre comparaison générale et comparaison spécifique. Au sein d'un compartiment, gardé par une vérification spécifique, seul des "checks" généraux sont effectués. Les "checks" spécifiques sont définis par ceux ne pouvant être déterminés par mutation d'un fuzzer.

**Driller**, commence par exécuter le fuzzer qui opère avec ou sans cas de tests initiaux (*avec étant préférable*). Celui-ci explore le premier compartiment jusqu'à atteindre le premier *complex-check* ayant des entrées spécifiques sur lequel il bloque. La solution à RQ\_COMB2 est que, lorsque le fuzzer a effectué un nombre prédéterminé de mutations – proportionnel à la longueur – sans avoir généré de nouvelles transitions d'états alors il est "coincé". L'exécuteur symbolique sélectif est alors invoqué sur le *complex-check* à partir des entrées "intéressantes". Une entrée est considérée comme intéressante si l'une des deux conditions suivantes est satisfaite :

- Le chemin induit par l'entrée était le premier à activer certaines transitions d'états ;
- Le chemin induit par l'entrée était le premier à être placé dans un *loop bucket* [1].

Ces conditions conservent un nombre réduit d'entrées transmises à l'exécuteur symbolique tout en conservant une bonne probabilité que les entrées mutées par DSE puissent mener à de nouveaux compartiments. Après cela, chaque entrée est tracée symboliquement pour identifier celles menant au *complex-check* que le fuzzing n'a pas réussi à satisfaire. La résolution de contrainte permet alors d'identifier des nouvelles entrées qui sont renvoyées au fuzzer qui explore le nouveau compartiment débloqué. Ce processus est répété jusqu'à ce qu'un crash soit découvert.

#### *loop bucket*

AFL détermine le nombre d'itérations de boucles ayant été exécutées et les compare avec les précédentes entrées étant passées par la même boucle. Ces chemins sont placés dans un *loop-bucket* en fonction du logarithme de leur nombre d'itérations. Seul un chemin par "panier" est considéré pour faire évoluer l'algorithme génétique. De cette manière seuls  $\log(n)$  chemins sont considérés pour chaque boucle (et non pas  $n$ ).

### 13.2.4 Implémentation

La combinaison utilise AFL [383] (cf. Section 10.2) en mode QEMU [26] pour se libérer de la dépendance au code source. Aucune modification n'a été effectuée sur le code d'AFL. L'exécution symbolique est effectuée par `angr`. Le code de `Driller` écrit en Python représente 1000 lignes de code et s'occupe donc principalement d'effectuer la glue entre les deux outils. Aucune mention n'est faite de modifications ou d'analyses dédiées implémentées dans `angr`, mais cela est très probablement le cas.

#### Note

`Driller` étant basé sur `angr` pour l'aspect symbolique, le modèle mémoire, la gestion des états et toutes les caractéristiques spécifiques au DSE sont celles de `angr` (cf. Section 7.5).

### 13.2.5 Conclusion

L'évaluation sur la base du CGC montre que `Driller` est capable de trouver 6 crashes de plus que l'union des résultats du fuzzing et de l'exécution symbolique indépendamment. Sans être exceptionnel, cela leur a permis de trouver autant de bugs que `Mayhem` lors des phases de qualification du CGC (cf. 12.2). Les auteurs justifient les résultats par le fait que les binaires sont compliqués et donc ce faible gain est tout de même significatif.

La notion de général et spécifique “checks” semble un peu artificielle et ressemble à une tautologie. En effet, cette distinction est présentée comme préexistante dans un programme, justifiant ainsi l'utilisation d'un fuzzer sur l'un et du DSE sur l'autre alors qu'en pratique, la distinction s'effectue lorsque le fuzzer n'arrive pas à résoudre un check. Cette frontière entre général et spécifique est donc toute relative aux performances du fuzzer.

Au-delà du concept de compartiment, la notion d’“entrées intéressantes” permet à l'algorithme d'exécution symbolique d'être réellement sélectif et donc de maximiser les chances de couvrir de nouvelles régions (*compartiment dans leur phraséologie*) du code.

L'inconvénient significatif de `Driller` est qu'il ne fonctionne que pour les binaires DE-CREE du CGC. Il peut théoriquement fonctionner pour d'autres formats, mais cela n'est pas documenté. Le faire fonctionner sur d'autres binaires nécessiterait de redévelopper la glue entre `angr` et AFL, qui est spécifique à DECREE.

## 13.3 Combinaison #2 : Angora

**Angora** est un fuzzer basé sur la mutation d'entrée, dont le but est d'explorer de nouveaux chemins en effectuant, non pas de l'exécution symbolique mais de la propagation de teinte. Cette approche le rend novateur et le place comme une combinaison, car il n'est ni simplement un fuzzer, ni simplement un outil d'exécution symbolique. L'outil est relativement jeune : le code source d'**Angora** en version 1.0.0 a été publié en décembre 2018<sup>145</sup>, peu après la publication du papier associé[71]. Depuis, le projet semble plutôt actif en termes de commits bien que ceux-ci correspondent pour la majorité à des correctifs de bugs. L'outil est écrit en Rust et malgré sa jeunesse, semble maintenu encore à l'heure de l'écriture de ce rapport.

```

ANGORA      (\_/_)
FUZZER      (=`o') .o

-- OVERVIEW --
TIMING | ALL: [00:00:00],   TRACK: [00:00:00]
COVERAGE | EDGE: 4.00,   DENSITY: 0.00%
EXECS   | TOTAL: 1,     ROUND: 1,     MAX_R: 0
SPEED    | PERIOD: 0.00r/s, TIME: 265.00us,
FOUND   | PATH: 1,     HANGS: 0,     CRASHES: 0

-- FUZZ --
EXPLORE | CONDS: 0, EXEC: 0, TIME: [00:00:00], FOUND: 0 - 0 - 0
EXPLOIT  | CONDS: 0, EXEC: 0, TIME: [00:00:00], FOUND: 0 - 0 - 0
CMPFN   | CONDS: 0, EXEC: 0, TIME: [00:00:00], FOUND: 0 - 0 - 0
LEN      | CONDS: 0, EXEC: 0, TIME: [00:00:00], FOUND: 0 - 0 - 0
AFL      | CONDS: 0, EXEC: 0, TIME: [00:00:00], FOUND: 0 - 0 - 0
OTHER   | CONDS: 0, EXEC: 1, TIME: [00:00:00], FOUND: 1 - 0 - 0

-- SEARCH --
SEARCH  | CMP: 0 / 0, BOOL: 0 / 0, SW: 0 / 0
UNDESIR | CMP: 0 / 0, BOOL: 0 / 0, SW: 0 / 0
ONEBYTE | CMP: 0 / 0, BOOL: 0 / 0, SW: 0 / 0
INCONIS | CMP: 0 / 0, BOOL: 0 / 0, SW: 0 / 0

-- STATE --
| NORMAL: 0d - 0p, NORMAL_END: 0d - 0p, ONE_BYTE: 0d - 0p
| DET: 0d - 0p, TIMEOUT: 0d - 0p, UNSOLVABLE: 0d - 0p

```

FIGURE 13.1 – Interface utilisateur - **Angora**

### 13.3.1 Introduction

**Angora** est un fuzzer dont la principale particularité est l'utilisation d'une *descente de gradient* (technique d'optimisation), afin de résoudre les contraintes de chemin. Contrairement aux techniques habituelles de fuzzing hybride, il n'utilise donc pas d'exécution symbolique. À la place, il introduit plusieurs techniques telles que la trace par teinte des entrées, l'analyse des branches contextuelles, et l'exploration de la longueur des entrées. L'outil fonctionne en deux phases distinctes. La première appelée phase *de tracking*, et la seconde appelée *boucle de fuzzing*.

La première phase de *tracking* permet à **Angora** de faire usage des différentes stratégies, décrites dans la section 13.3.2 et ainsi explorer des chemins parfois difficiles d'accès à un fuzzer simple.

145. <https://github.com/AngoraFuzzer/Angora>

La seconde phase, est une phase de fuzzing classique. En effet, **Angora** a ré-implémenté les méthodes de mutation standards d'**AFL** afin d'explorer rapidement les chemins faciles d'accès. Néanmoins, il est aussi possible d'utiliser une vraie instance d'**AFL** tournant en parallèle de la phase de *tracking*.

La combinaison d'une phase de fuzzing standard à une phase d'exploration de chemin moderne permet de découvrir efficacement de nouveaux chemins dans la cible, et ainsi atteindre des bugs complexes.

#### Note

L'instrumentation injectée à la compilation par les deux phases est différente. **Angora** nécessite de compiler deux exécutables différents pour chacune des phases.

### 13.3.2 Fonctionnalités

**Angora** emploie des techniques d'exploration de chemin modernes, permettant d'explorer la cible efficacement. Ces fonctionnalités sont décrites ci-dessous.

#### Analyse contextuelle et comptage des branches

Compter les instructions relatives au branchement permet d'obtenir des informations précises sur la couverture de code. Cette méthode est relativement efficace et est utilisée notamment par **AFL**. Cependant, cela ne permet pas de distinguer l'exécution d'une branche dans différents contextes. Le problème de cette non-distinction réside dans l'absence de connaissance de l'état interne du programme. En effet, certains bugs peuvent résider dans du code, à priori non vulnérable. Cependant, certains chemins spécifiques pourraient altérer le bon comportement attendu par ce code, et ainsi résulter en un bug. C'est cette problématique qu'**Angora** tente de résoudre via la méthode d'analyse contextuelle et de comptage des branches. En incorporant le contexte appelant, **Angora** est capable de détecter des exécutions subséquentes d'une même branche, qui pourraient potentiellement déclencher de nouveaux états de la pile d'appels.

**Angora** définit une branche comme un triplet  $(l_{prev}, l_{cur}, context)$  où  $l_{prev}$  est l'identifiant du basic-block précédent,  $l_{cur}$  l'identifiant du basic-block suivant et  $context$  le hash de la pile d'appels. Grâce au hashé de la pile d'appels contenu dans  $context$ , les branches identifiées par **Angora** permettent de distinguer facilement deux chemins différents menant au même code.

#### Teinte des entrées

**Angora** ne mute pas l'intégralité des entrées à chaque exécution. En effet, l'outil se base sur l'hypothèse que la majorité du code conditionnel dépend uniquement d'un nombre

Note

Ajouter un contexte à la définition d'une branche augmente radicalement le nombre de branches identifiées par l'outil. Cela peut parfois devenir problématique, comme dans le cas d'une fonction récursive. Afin de pallier le problème, la fonction de hash utilisée par **Angora** effectue au préalable, un XOR des identifiants des fonctions de la pile d'appels, permettant ainsi de facilement éliminer les redondances dues aux boucles.

limité d'octets de l'entrée. Il permet d'identifier, à l'exécution et grâce à un algorithme de teinte, les octets responsables du code conditionnel. **Angora** appliquera principalement des mutations sur ces derniers.

### Résolution de contraintes via descente de gradient

La fonctionnalité principale d'**Angora**, qui permet de le distinguer des autres outils, est sa capacité à résoudre des contraintes sans utiliser de moteur d'exécution symbolique, à qui on reproche souvent d'être trop couteux en temps. En effet, **Angora** se base sur la méthode d'optimisation de la **descente de gradient**. **Angora** considère les branches comme une contrainte, représentée par la fonction en boîte noire  $f(x)$ , où  $x$  est un vecteur de valeurs de l'entrée, ayant un impact sur le prédicat. De cette base, **Angora** évalue la branche en analysant les paramètres afin de déterminer quels octets de l'entrée ont une incidence, et le type de ces derniers (voir section 13.3.2).  $f()$  capture le calcul du chemin depuis le début de l'exécution jusqu'au prédicat actuel. La fonction sera alors catégorisée dans l'une des trois contraintes suivantes :

- $f(x) < 0$
- $f(x) \leq 0$
- $f(x) = 0$

Si le prédicat est constitué de plusieurs opérateurs logiques, l'outil le divisera alors en plusieurs propositions conditionnelles.

Note

**Angora** peut transformer n'importe laquelle de ces contraintes en une autre, pour faciliter les calculs.

Grâce à la modélisation de ces prédicats sous forme de contrainte, **Angora** peut alors utiliser la méthode de **descente de gradient** afin de les résoudre. Cependant, cette technique est généralement employée lorsqu'on dispose de la forme analytique de la fonction à analyser, ce qui n'est pas le cas ici. De plus,  $f(x)$  est censée être considérée comme une fonction continue, mais dans le cadre du fuzzing, cette fonction est discrète. En effet, la majorité des variables dans un programme sont discrètes, les éléments composant  $x$  sont donc eux aussi discrets. **Angora** tente de résoudre ces problèmes en utilisant une approximation numérique des valeurs.

**Note**

En théorie, la descente de gradient est capable de résoudre toutes les contraintes (si une solution existe), là où les solveurs de SMT peuvent présenter des difficultés face à des contraintes non linéaires par exemple. En pratique, la vitesse de résolution de la descente de gradient dépend de la complexité mathématique de la fonction.

**Inférence de type et de forme**

Afin d'appliquer l'algorithme de descente de gradient, il est nécessaire de résoudre certains problèmes, notamment, déterminer la taille des paramètres de la fonction  $f()$  et d'inférer leur type. **Angora** doit déterminer dans un premier temps les octets de l'entrée qui sont utilisés conjointement par le programme pour ensuite inférer le type de cette valeur. **Angora** appelle le premier problème **inférence de forme**, tandis que le second est catégorisé **d'inférence de type**. Ces problèmes sont résolus dynamiquement grâce à de l'analyse par teinte.

Pendant la phase d'analyse par teinte, lorsqu'une instruction lit une séquence d'octets depuis l'entrée, **Angora** identifie la taille de la primitive (1, 2, 4 ou 8 octets), puis marque les octets comme appartenant à la primitive de taille. Dans le cas où plusieurs instructions traitant ces données sont utilisées avec des primitives de tailles différentes, alors **Angora** sélectionne alors la taille minimale.

Pour l'inférence du type, **Angora** se repose sur la sémantique de l'instruction qui opère sur cette valeur. Par exemple, si une instruction opère sur un entier signé, alors **Angora** pourra inférer que l'opérande correspondante est un entier signé. Lorsqu'une valeur est utilisée indifféremment en tant qu'entier signé ou non signé, l'outil traite alors la valeur comme non signée.

**Note**

Lorsqu'**Angora** ne parvient pas à inférer précisément la taille et le type d'une valeur, la résolution de contrainte via descente de gradient reste valide. Elle sera juste plus longue et moins optimisée.

**Découverte de la taille de l'entrée**

Certains bugs ne peuvent être déclenchés qu'à partir du moment où la taille de l'entrée a dépassé un seuil. Cela crée un dilemme sur la décision de la taille de l'entrée qu'**Angora** tente de résoudre. L'outil instrumente le code, de manière à détecter les cas où une entrée plus longue permettrait d'explorer de nouvelles branches, et détermine aussi la taille minimum de l'entrée requise. **Angora** n'augmente la taille de l'entrée que lorsqu'il détermine que cela est nécessaire pour explorer de nouvelles branches.

Note

Lors de tests réalisés afin d'évaluer l'outil pour le projet PASTIS, mais aussi lors de tests internes, la méthode d'exploration de la longueur des entrées semblait parfois inefficace, et ne permettait pas de muter une entrée avec une taille supérieure à l'entrée la plus longue du corpus initial.

Lors de l'exécution teintée, **Angora** identifie l'espace mémoire de destination, où une fonction de lecture ira stocker le contenu de l'entrée. Il marque aussi la valeur de retour de la fonction de lecture avec un label. Si la valeur de retour est utilisée dans une condition, et que la contrainte n'est pas satisfaite, **Angora** augmente alors la taille de l'entrée de manière à satisfaire la condition et la fonction de lecture.

Cette technique n'est pas infaillible, en effet, il existe des cas où ce mécanisme ne fonctionne pas, **Angora** prévoit de mettre à jour les mécanismes de détection au fur et à mesure de leur découverte.

### 13.3.3 Instrumentation

**Angora** se base sur LLVM pour les passes d'instrumentation du programme. Le code d'instrumentation permet de :

- Collecter des informations basiques sur les opérations conditionnelles.
- Enregistrer les traces d'exécution afin d'identifier de nouvelles entrées.
- Fournir un contexte d'informations pour le comptage des branches à l'exécution.
- Recueillir les valeurs des expressions dans les prédicats.

Outre la collecte d'informations, **Angora** lie aussi les opérations conditionnelles aux octets relatifs de l'entrée lors de l'analyse par teinte.

Comme décrit dans l'introduction, **Angora** fonctionne en deux phases : la 1) phase de tracking et 2) la phase de fuzzing. L'instrumentation étant différente pour ces deux phases, l'outil compile deux versions de l'exécutable, une par phase, à des fins d'efficacité.

Note

**Angora** supporte uniquement l'architecture `x86_64` et se base sur une version relativement ancienne de LLVM (4).

### 13.3.4 Optimisations

**Angora** est optimisé avec des techniques similaires à AFL, telles que le `fork server` et le *CPU binding*, permettant une initialisation des processus cibles rapide.

De plus, **Angora** propose une fonctionnalité permettant de reconnaître les chaînes de caractères et la comparaison de tableaux dans les opérations conditionnelles. Par exemple, il transforme les appels à `strcmp(x, y)` par un opérateur spécial `strcmp` donnant l'opération `x strcmp y`. Cet opérateur est compris par **Angora** et permet notamment de casser les comparaisons des chaînes de caractères.

### 13.3.5 Conclusion

**Angora** propose des fonctionnalités novatrices et uniques en leur genre. Les résultats obtenus à l'aide de cet outil sont étonnamment bons. Le projet reste aujourd'hui maintenu et ses évolutions méritent de garder un œil sur l'activité du projet. Cependant, l'outil reste jeune et n'a pas été éprouvé. **Angora** reste restreint à l'architecture `x86_64`, et présente quelques points d'améliorations tels que :

- Mettre à jour la version de Clang/LLVM ;
- Fixer les bugs relatifs à la mutation de la taille de l'entrée ;
- Proposer une interface plus adaptée.

## 13.4 Combinaison #3 : Qsym

**Qsym** est un outil d'exécution concolique combiné à une instrumentation basée sur Pin de Intel. Il est conçu pour être combiné facilement avec un fuzzer tel qu'AFL et ainsi proposer des capacités de fuzzing hybride. Ces travaux de recherche ont été effectués par le laboratoire *Systems Software & Security Lab* (SSLab) de l'université Georgia Institute of Technology<sup>146</sup> (connu comme *GaTech*). La seule et unique publication disponible au sujet de **Qsym** est la suivante :

*“QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing” par Insu Yun and Sangho Lee and Meng Xu and Yeongjin Jang and Taesoo Kim publié au 27th USENIX Security Symposium 2018*

La conférence USENIX a aussi mis en ligne les slides<sup>147</sup> et la vidéo<sup>148</sup> de la présentation de l'article. Le code, essentiellement du C++ pour Pin et du Python pour la combinaison avec AFL, a été publié en août à l'issue de la conférence, sous la licence de *The Regents of the University of California*. Celui-ci n'a pas été mis à jour depuis décembre 2018. Le développement sur le projet github<sup>149</sup> est donc essentiellement au point mort. La version analysée est la 0.1 correspondant au commit 6f00c3d84835e4dc3afbfde56e12540b4b44c0b7.

Les résultats de la publication [380] montrent que **Qsym** se révèle particulièrement efficace. En effet, combiné avec AFL il permet d'augmenter la couverture de code significativement, mais aussi d'atteindre beaucoup plus de bugs (et ce plus efficacement) sur des bases de référence comme LAVA [109].

### 13.4.1 Problématique

Parmi les approches de fuzzing hybride les auteurs énoncent les trois problèmes ayant un impact sur les performances :

**P1** : l'exécution symbolique est lente

- les DSE actuels utilisent une teinte “gros-grain” au niveau basic-block et exécutent donc symboliquement toutes les instructions d'un block (*en particulier Driller* [333]), alors qu'en moyenne seuls 30% des instructions d'un basic-block nécessitent d'être exécutées.
- les DSE purement symboliques qui émulent tout l'OS (*et qui ne sont donc pas concrets*) sont beaucoup trop coûteux en temps et ne passent pas à l'échelle (*e.g angr* [320], *Manticore* [33]);

**P2** : Le mécanisme de snapshot des états ralentit l'exécution (cf.12.2) ;

**P3** : L'inflexible correction de l'exécution symbolique impose des contraintes non nécessaires à la génération d'entrées.

---

146. <https://gts3.org>

147. [https://www.usenix.org/sites/default/files/conference/protected-files/usesec18\\_slides\\_yun.pdf](https://www.usenix.org/sites/default/files/conference/protected-files/usesec18_slides_yun.pdf)

148. <https://www.youtube.com/watch?v=cXr1ZXp40jA>

149. <https://github.com/sslabs-gatech/qsym>

### 13.4.2 Approche

Afin de résoudre certaines de ces problématiques, **Qsym** tente de relaxer (relâcher) la correction *soundness* des moteurs d'exécution concolique standards. Cette approche n'est pas nouvelle [117, 288], mais elle se démarque de la combinaison usuelle fuzzer/DSE avec une correction stricte et permet d'améliorer significativement les performances. Les auteurs affirment que cette perte de correction n'est pas problématique puisqu'il est possible de vérifier rapidement si les entrées générées sont valides. Pour cela, **Qsym** intègre l'exécution symbolique au cœur de l'exécution native grâce à de la translation dynamique de binaires ([Dynamic-Binary-Translation \(DBT\)](#)).

#### Note

Le [DBT](#) est une technique d'instrumentation dynamique qui se base sur le désassemblage de chaque basic-block, les modifient au besoin, puis les exécutent via du [JIT](#). Elle s'oppose à une instrumentation avec debugger nécessairement moins polyvalente.

**Qsym** n'utilise pas l'habituelle couche de traduction dans un langage intermédiaire qui permet d'obtenir de meilleures performances, au coût d'un développement plus complexe. De plus, **Qsym** optimise les répétitions de tests concoliques : il supprime le mécanisme d'instantané de l'état du programme, ce qui a un coût non négligeable dans le fuzzing hybride usuel. Les développeurs ont opté pour une exécution concrète afin de modéliser l'environnement extérieur. Les contraintes ainsi collectées par **Qsym** ne sont pas complètes, et ne permettent de résoudre qu'une portion des contraintes relatives à l'exploration d'un chemin. Cependant, cela est plus efficace pour les chemins présentant des contraintes importantes. Enfin **Qsym** n'exécute symboliquement qu'une partie des instructions, permettant une exécution plus rapide que les outils de fuzzing hybride usuels (cf.[P2](#)). Toutes ces optimisations facilitant le passage à l'échelle sont détaillées dans la section suivante.

### 13.4.3 Optimisations

#### Exécution symbolique granulaire (instruction)

**Qsym** exécute symboliquement seulement un ensemble réduit d'instructions, celles nécessaires à la génération des contraintes symboliques. À l'inverse de [angr](#) effectuant une exécution dont la granularité minimale est celle d'un basic-block, **Qsym** se positionne au niveau instruction. Via un outil d'instrumentation dynamique de binaire, **Qsym** exécute à la fois des instructions natives et des instructions symbolisées. Ces opérations opérant dans un seul processus, il est relativement rapide de changer d'un mode à l'autre (l'équivalent d'un appel de fonction).

## Résolutions de contraintes optimistes

`Qsym` s'évertue à générer de nouveaux cas de tests à partir des contraintes générées en choisissant de manière optimiste des portions de celle-ci, si elle n'est pas solvable en entier (*Optimistic Solving*). Sélectionner ainsi de manière opportuniste des morceaux du prédicat complet permet de générer des entrées valides pour le chemin. L'algorithme élimine d'abord les contraintes qui ne sont pas liées à la dernière contrainte du chemin et donc, élimine les contraintes qui n'impactent pas le résultat de la résolution (*mais alourdisseront la formule*).

## Ré-exécution face à instantané

Afin de répéter les tests, le moteur d'exécution concolique de `Qsym` favorise la ré-exécution de la cible plutôt que la conservation d'un instantané (*snapshot*) de celle-ci. Il est en général très coûteux pour un outil d'exécution symbolique d'atteindre le chemin d'exécution souhaité en redémarrant le programme. Bien que l'approche de l'instantané soit adaptée aux moteurs d'exécution symbolique, l'instantané a un coût non négligeable en temps qui n'est généralement pas un goulot d'étranglement. Cependant, dans le cas de `Qsym`, le moteur d'exécution étant particulièrement rapide, il est relativement simple d'atteindre le chemin souhaité via une exécution complète. L'intérêt de l'instantané devient moindre, et pourrait même impacter négativement les performances de `Qsym` (cf. **P3**).

## Élagage de basic-block

Les auteurs de `Qsym` ont observé que les contraintes répétitives générées par le même code ne sont, généralement, pas utiles à l'amélioration de la couverture de code. Plus précisément, les contraintes générées par les opérations de calcul intensives dans un programme sont rarement solvables (non-linéaires). D'après ce constat, `Qsym` tente de détecter les basic-blocks répétitifs et les éague de l'exécution symbolique afin de ne générer qu'un ensemble réduit de contraintes. `Qsym` éague les basic-blocks en les catégorisant par fréquence. L'outil éague tous les basic-blocks ayant été exécutés un trop grand nombre de fois en suivant un algorithme de recul exponentiel *exponential back-off* pour décider ou non d'éaggerer un block. Cette proposition est fortement contraignante, et pourrait ainsi éaggerer trop de basic-block. Afin d'éviter ce comportement, `Qsym` emploie deux méthodes.

*Grouping multiple execution.* La première méthode consiste à regrouper les exécutions, afin de minimiser l'élagage de basic-block. `Qsym` détermine au préalable une taille de groupe (8 dans le papier). Cette taille est une limite à partir de laquelle la fréquence de rencontre du basic-block est incrémentée. Le compteur est donc incrémenté toutes les huit occurrences. Les auteurs exposent cette méthode comme aidant à conserver les contraintes essentielles à la découverte de nouveaux chemins, sans affecter l'exécution symbolique. Une valeur faible de taille de groupe permet de s'assurer que les contraintes générées ne seront pas trop complexes à résoudre.

*Context-sensitivity.* La seconde méthode proposée par **Qsym** est la sensibilité au contexte d'exécution. Cette méthode est similaire à celle employée par **Angora** 13.3.2. Elle permet de distinguer l'exécution d'un basic-block qui aurait lieu dans un contexte d'exécution différent. **Qsym** maintient une pile d'appels de l'exécution courante, qu'il « hashe » afin de différencier les contextes d'exécution.

### 13.4.4 Implémentation

**Qsym** se base sur l'outil Intel Pin comme outil d'instrumentation dynamique, et se sert de l'outil `libdft` [204] pour la teinte des entrées, et la gestion des appels système sous les environnements 64 bits. La version de `Pin` est relativement ancienne (2.14), celle-ci ayant été publiée le 3 Février 2015.

Il propose un fuzzer hybride reposant sur AFL en tant qu'outil de démonstration, mais celui-ci peut facilement être interchangé, car tous les échanges ont lieu via le dossier de travail d'AFL. Le code de glue entre les deux représente 565 lignes de code Python. Le tableau 13.2 donne le nombre de lignes de code par composants du projet :

Composant	Lignes de code
Exécution concolique (core)	12528 Loc C++
Génération des expressions	1913 Loc C++
Abstraction des appels système	1577 Loc C++
Fuzzing hybride	565 Loc Python

TABLE 13.2 – Lignes de code des composants de **Qsym**

Dans la démonstration, deux instances d'AFL sont exécutées, la première est l'instance maître, et la seconde est l'esclave. **Qsym** fonctionne en parallèle de ces exécutions, et analyse en détail chacun des fichiers dans la queue de l'instance esclave, et génère des entrées pour le corpus. Le fonctionnement parallèle de **Qsym** rend particulièrement facile son intégration avec un fuzzer.

### 13.4.5 Conclusion

Les résultats de l'article sont édifiants. Celui-ci trouve 14 fois plus de bugs que **VUzzer** [290] sur LAVA-M et plus de bugs dans 104 des 126 binaires du CGC par rapport à **Driller** [333]. En plus de cela, 13 vulnérabilités précédemment inconnues ont été trouvées dans des utilitaires Debian dont `tcpdump`, `ffmpeg`, `objdump`, `openjpeg` etc.

**Qsym** introduit de nombreuses optimisations en capitalisant les observations faites des précédentes approches et outils. L'évaluation de **Qsym** montre qu'il améliore drastiquement les performances d'AFL. Certains choix de design comme celui de supprimer l'IR permet certes de gagner en performances, mais réduit considérablement la réutilisabilité

et la portabilité du code. Par exemple, l'outil ne supporte aujourd'hui que l'architecture **x86\_64**, un portage sur d'autres architectures s'avèrerait pénible et coûteux en temps en l'état.

Au-delà des performances, un avantage majeur de **Qsym** est sa capacité à se combiner facilement avec d'autres fuzzers, grâce à l'API Python qu'il propose. Tout comme **Angora**, même si l'efficacité de l'outil est réelle, sa jeunesse et ses perspectives d'évolution rendraient floue la pérennité d'une combinaison basée sur ce dernier.

## 13.5 Combinaison #4 : cyberdyne

cyberdyne est le nom du CRS utilisé par Trail of Bits pour la compétition du Cyber Grand Challenge. Bien qu'il n'eût pas été possible de le tester, cet outil a été choisi, car la majorité de ses composants ont été « open-sourcé ». Avec l'équipe Shellphish ce sont les deux seuls concurrents à avoir adopté cette démarche. De plus le fonctionnement de leur CRS est relativement bien documenté dans divers articles de blog<sup>150 151</sup>, articles scientifiques [142], publications de conférences comme InfiltrateCon [104] ou encore CounterMeasure [143] et leurs deux vidéos associées<sup>152 153</sup>. Cette étude agrège les différentes informations distillées dans ces ressources, mais se focalise principalement sur la publication scientifique :

*“The Past, Present, and Future of Cyberdyne” par Petter Goodman, Artem Dinaburg publié au IEEE Symposium on Security and Privacy, S&P 2018*

Comme la plupart des concurrents du CGC, cyberdyne mélange fuzzing et exécution symbolique.

### 13.5.1 Architecture

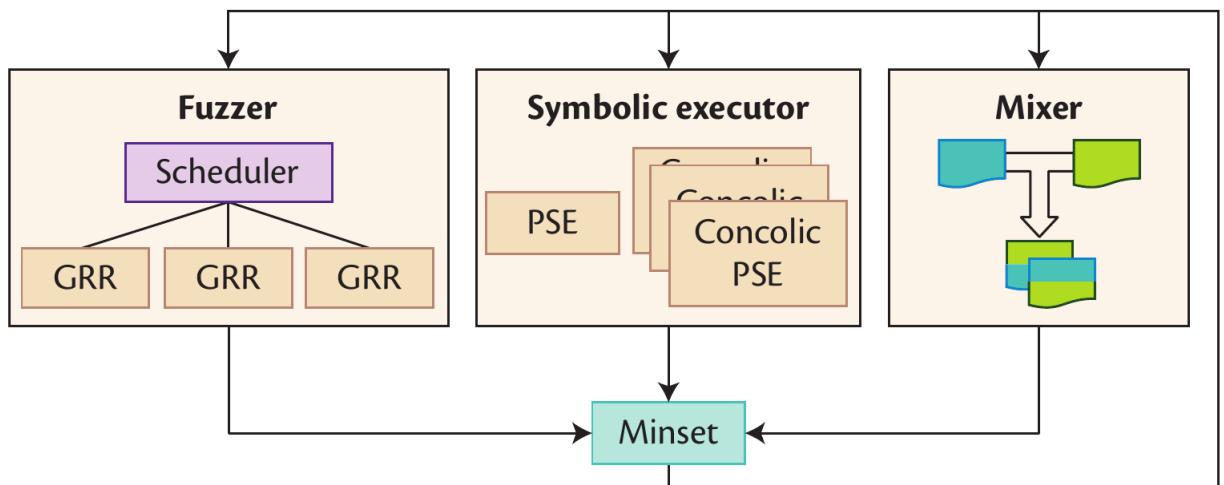


FIGURE 13.2 – Architecture des composants de cyberdyne (src. [142])

L'architecture de ce CRS s'articule autour des quatre composants suivants fonctionnant en autonomie :

150. <https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge>  
 151. <https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems/>

152. <https://vimeo.com/180882544>

153. <https://www.youtube.com/watch?v=ugMd3-yea40>

- le fuzzer (GRR) ;
- l'exécuteur symbolique ;
- le mixer effectuant le mélange des entrées issues des deux précédentes sources ;
- le minset implémentant la fonction de *fitness* pour choisir la future configuration à tester basée sur l'aptitude de l'entrée à couvrir de nouvelles portions de code.

La figure 13.2 dresse le schéma des interactions entre ces différents composants. Le mixer prend les entrées favorisant les meilleures couvertures et utilise plusieurs heuristiques (*non décrites*) pour générer de nouvelles entrées candidates.

### 13.5.2 Fuzzing



FIGURE 13.3 – Logo GRR

La partie fuzzing s'articule en deux composants le **scheduler** et GRR le fuzzer. Le *scheduler* assure le partage de la charge CPU entre les différentes instances du fuzzer. C'est aussi celui-ci qui reçoit les entrées du *minset* et les envoie aux différentes instances du fuzzer. Le deuxième composant est GRR<sup>154</sup> un fuzzer qui, à l'inverse des autres concurrents du CGC, est conçu de toute pièce. Celui-ci fait de la translation binaire dynamique DBT de x86 et x86-64. Il se base sur XED pour le désassemblage des instructions et l'instrumentation dynamique est issue de Granary<sup>155</sup> développé par le même auteur.

Donc GRR est une spécialisation de Granary pour du fuzzing. Enfin GRR s'appuie sur Radamsa [172] pour la mutation des entrées.

#### Fonctionnalités

Les principales fonctionnalités sont les suivantes :

- Cache du code “translate” à travers les exécutions dans un cache en mémoire<sup>156</sup>. Cela permet d'obtenir une sorte de “forkserver”, mais à base de DBT (et non pas d'instrumentation statique comme avec AFL).
- implémentation des *snapshot* (pour aussi “forker des exécutions”).
- Cache de code persistant.
- Instrumentation multi-process, c'est à dire permet à plusieurs programmes 32-bit d'être instrumenté dans le même espace mémoire 64-bit.
- Mutations simples implémentées dans GRR (bit-flip) et support de Radamsa pour les mutations complexes.
- Support du code auto-modifiant (e.g compilateur JIT etc).

154. <https://github.com/trailofbits/grr>

155. <https://github.com/Granary/granary>

156. Pin par exemple réinstrumente à chaque fois, car il est plutôt conçu pour instrumenter des programmes à exécution longue

- Émulation des appels systèmes et I/O pour les programmes DECREE.

En interne, cet outil utilise plusieurs spécificités de DECREE pour optimiser le fuzzing de ces programmes. En effet ceux-ci sont 32bits donc **GRR** tire profit des registres 64bits inutilisés par le programme pour stocker des informations relatives à l'instrumentation.

Étant spécialisé pour le CGC, **GRR** émule complètement les 7 appels système de DECREE et donc tout ce qui est interaction avec le réseau, le système de fichier, etc. Cela va à l'encontre de l'approche traditionnelle d'un fuzzer qui normalement exécute concrètement tout le programme. L'avantage de cette approche est qu'elle permet une stricte reproductibilité de l'exécution en éliminant toute source de non-déterminisme.

### 13.5.3 Exécution symbolique

L'architecture de **cyberdyne** a la particularité de permettre à deux DSE de fonctionner indépendamment l'un de l'autre. Une ancienne version utilisait **PySymEmu** (PSE) et **KLEE** dont le LLVM "lifté" provenait de **McSema** [36] (cf. Section 6.3.3). Or, ce dernier a fini par être abandonné, car **KLEE** effectue les appels système sur l'OS lorsqu'il ne possède pas de modèle symbolique. Ces effets de bord était manifestement délétère par rapport à l'approche purement "émulée" de **GRR** et **PySymEmu**. De plus il semblerait que **KLEE** fonctionne assez mal sur LLVM reconstruit depuis le binaire et qu'il soit à l'inverse optimisé pour fonctionner sur le code généré par **clang**.



FIGURE 13.4 – Logo PySymEmu

**PySymEmu** est quant à lui l'ancêtre de **Manticore** mais spécifique à DECREE. Il partage une partie des fonctionnalités décrites en Section 7.3. Sa principale force est son aptitude à utiliser les *snapshots* générés par **GRR** pour effectuer l'exécution symbolique.

### 13.5.4 Minset

Ce composant implémente la fonction de *fitness* sous forme d'algorithme génétique favorisant les entrées maximisant la couverture (comme **AFL**). Les entrées ayant précédemment découvert de nouvelles portions de code sont donc favorisées. L'algorithme utilise un calcul étendu de la couverture de code. Avant chaque exécution de branche, il enregistre un 3-uplet d'adresses correspondant à :

- l'adresse de la branche la plus récemment exécutée ;
- l'adresse de l'instruction de branchemen (effectuant le saut) ;
- l'adresse (distant) sur le point d'être exécutée.

Les sauts indirects comme les **switch**, les vtables, et les retours d'instruction comptent aussi comme des branches dans l'algorithme.

### 13.5.5 Conclusion

Cet outil se targue d'avoir été le premier outil utilisé pour faire de l'audit commercial automatisé. Ce fut celui de `zlib` pour le fond *Secure Open Source* (SOS) financé par Mozilla. Cet audit fut réalisé conjointement avec TrustInSoft<sup>157</sup> développant et commercialisant un analyseur statique basé sur `Frama-C` [209]. Le rapport [105] indique que 5 vulnérabilités ont été trouvées dont une de sévérité “médium”. Malheureusement il est difficile d'évaluer l'implication de chacun des deux outils dans la découverte de ces bugs.

Le score de `cyberdyne` au CGC (9<sup>ème</sup>) est tout à fait honorable, et l'humilité des explications fournies concernant leur échec est fortement appréciable, car communiquer sur des échecs est au moins aussi important que communiquer sur des succès.

L'architecture complète de `cyberdyne`, bien que non disponible, donne un aperçu intéressant de ce qu'il est possible de faire au niveau fuzzing et DSE pour une combinaison. Externaliser comme un programme indépendant le mixer et le minset est particulièrement intéressant. Ainsi, bien que la plupart des outils soient dédiés au challenge CGC et au Linux DECREE, ils fournissent une vue intéressante en vue de créer une combinaison pour le projet PASTIS.

---

157. <https://trust-in-soft.com/>

## 13.6 Combinaison #5 : Eclipser

Le choix d’analyser **Eclipser** est non seulement motivé par la disponibilité de l’outil<sup>158</sup> mais aussi par l’originalité de l’approche et la nouveauté des résultats. En effet ces travaux sont les plus récents de l’état de l’art puisqu’ils n’ont même pas encore été publiés : ils le seront à ICSE à Montréal fin Mai 2019. La version analysée est le prototype 0.1 posté sur le Github du projet. Ces travaux sont dirigés par Sang Kil Cha (*ayant travaillé sur Mayhem*) au laboratoire SoftSec du KAIST<sup>159</sup> à Daejeon en Corée du Sud. L’article scientifique étudié est :

“Grey-box Concolic Testing on Binary Code” par Jaeseung Choi, Joonun Jang, Choongwoo Han et Sang Kil Cha publié au 41th International Conference on Software Engineering, ICSE 2019

Tous les artefacts des expériences effectuées sont aussi fournis<sup>160</sup> ce qui permet une reproductibilité parfaite et une transparence des résultats.

Dans la continuité de **Angora** [71] et **Qsym** [380] qui cherchent à se défaire de la rigidité du DSE, **Eclipser** propose une combinaison *grey-box* d’exécution concolique se passant complètement de la résolution de formule par SMT (l’un des principaux goulets d’étranglement). Il se distingue donc de **Angora** qui se base sur de la teinte et de **Qsym** ayant quand même recourt à de la résolution de formules.

### 13.6.1 Approche

**Eclipser** se présente comme un outil de test concolique *grey-box* à l’intersection entre fuzzing whitebox et fuzzing *grey-box*. L’idée est de générer des cas de tests satisfaisant des conditions de branchement comme en fuzzing whitebox tout en gardant la simplicité du fuzzing *grey-box* (*et donc ne s’appuie pas sur de lourdes analyses statiques, de teinte, etc.*).

Elle se rapproche de la recherche générationnelle des fuzzer whitebox (cf. Section 12.3) où, une seule exécution du programme produit un ensemble de cas de tests correspondant à chaque condition de branchement rencontrée lors de l’exécution. Cependant, cette approche se différencie du fuzzing whitebox, car comme **Qsym** [380], il utilise une forme “approximée” (relaxée) des contraintes qui ne décrivent que partiellement les conditions permettant d’exercer le chemin (*favorisant la simplicité au dépens de la précision*). L’approche est dite *search-based*. Les travaux récents sur **SBF** [337] proposent aussi cette approche pour effectuer du fuzzing.

---

158. <https://github.com/SoftSec-KAIST/Eclipser>

159. <https://softsec.kaist.ac.kr/>

160. <https://github.com/SoftSec-KAIST/Eclipser-Artifact>

### 13.6.2 Problématique

L'article identifie le problème du critère de couverture utilisé par la fonction de *fitness* qui détermine et ordonne les configurations qui vont être fuzzées. Ce problème commun au *coverage-based grey-box fuzzers* est dû à *l'insensibilité insensitivity* de cette fonction. Autrement dit, cette fonction est incapable de déterminer quelle entrée permet de satisfaire une condition de branchement donnée. Toutes les fonctions utilisées (couverture de basic-block, couverture de branches) sont insensibles, car il n'y a pas de fonction de *fitness* intermédiaire effectuant le lien entre deux exécutions prenant respectivement un branchement et sa négation. Une telle fonction est la fonction de distance de branche *branch distance* [213, 346] (notamment utilisée par [Angora](#) [71]). [Eclipser](#) utilise une idée similaire pour inférer et résoudre directement des conditions de branchement approximées.

### 13.6.3 Test concolique *grey-box*

L'aspect clé de l'approche est de maintenir un sous-ensemble de contraintes approximées pour chaque octet de l'entrée. Cet ensemble noté  $pc$  est un dictionnaire associant à chaque octet de l'entrée la contrainte qui lui est associé. Ces contraintes sont en fait un intervalle de valeurs (*et non pas des termes SMT comme le mot laisse suggérer*). L'algorithme définit les fonctions suivantes et l'algorithme est donné en Figure 3 :

- $\text{Spawn}(p, s_p, k) \rightarrow execs$  : Génère  $N_{spawn}$  nouvelles entrées distinctes en modifiant le  $k^{\text{ème}}$  octet de  $s_p$  satisfaisant la contrainte  $pc[k]$  et produit les exécutions  $execs$  sur  $p$  ;
- $\text{Identify}(p, execs) \rightarrow cond$  : identifie les conditions  $cond$  affectées par le  $k^{\text{ème}}$  octet muté dans les exécutions produites  $execs$  ;
- $\text{Select}(conds) \rightarrow cond'$  : renvoie un sous-ensemble aléatoire de conditions dans  $conds$  de  $N_{solve}$  éléments (tout en conservant leurs ordres d'apparition) ;
- $\text{Search}(p, k, pc, execs, cond) \rightarrow s'_p, c$  : Pour chaque  $cond$  tel que  $cond \in cond$ , essaye de pénétrer  $cond$  en générant une nouvelle entrée  $s'_p$  exerçant la négation de  $cond$  (et renvoie  $c$  la contrainte sur les entrées pour couvrir  $cond$ ).

---

**Algorithm 3** Algorithme de test concolique *grey-box*

---

```

1: function GREYCONC( $p, s_p, k$ )
2:    $pc \leftarrow \{\}$ 
3:   seeds  $\leftarrow \emptyset$ 
4:   execs  $\leftarrow \text{Spawn } (p, s_p, k)$ 
5:   cond  $\leftarrow \text{Identify } (p, execs)$ 
6:   for cond in Select (conds) do
7:      $s'_p, c \leftarrow \text{Search } (p, k, execs, cond)$ 
8:     seeds  $\leftarrow$  seeds +  $s'_p$ 
9:      $pc \leftarrow pc \wedge c$ 
10:   end for
11:   return seeds
12: end function

```

---

`Spawn` mute  $k$  fois l'entrée  $s_p$  en satisfaisant la contrainte  $pc[k]$  qui, étant un intervalle, revient à choisir une valeur dans cet intervalle. Cette contrainte étant approximée, il est possible que la valeur générée ne permette pas de couvrir en pratique le chemin courant.

`Search` résout une condition de branchement  $cond$  pour en couvrir une nouvelle. Il produit une nouvelle graine  $s'_p$  pour couvrir la nouvelle branche et ajoute  $c$  la condition de branchement pour continuer à couvrir le chemin courant. L'intuition est d'établir la relation entre l'octet et l'opérande de la condition de branchement (cf. problématique 13.6.2) pour en déduire une solution potentielle. Ainsi, `Search` cherche à inférer des relations *linéaires* ( $y = a \times x + b$ ) ou *monotones* ( $a = b$ ) entre les exécutions *execs* et les variations de l'octet  $k$  les ayant produites (cette approche ayant déjà été explorée par le passé [223, 368]). Ce design se base sur l'observation empirique que *la majorité des branchements conditionnels ont tendance à avoir des contraintes monotones ou linéaires*. Cette inférence ne se borne pas à un octet, mais tente l'inférence en utilisant les octets voisins ( $k-8$ , à  $k+1$ ) dans le cas où le  $k^{\text{ème}}$  octet est un fragment d'entier 32 ou 64 bits par exemple. A noter que si `Search` ne trouve pas de solution une mutation aléatoire est appliquée pour compenser.

#### Note

Un exemple de relation linéaire sur une condition de branchement est l'exemple d'un `switch` dont l'adresse sera souvent de la forme  $y = ith * 8 + base$  où  $ith$  est le  $i^{\text{ème}}$  élément dans la table de saut ayant pour base l'adresse *base* ( $8$  étant la taille d'une adresse 64 bits).

### 13.6.4 Architecture de Eclipser

L'architecture complète combine la fonctionnalité test concolic *grey-box* présentée ci-dessus avec du fuzzing *grey-box* plus traditionnel permettant notamment de traverser les conditions non linéaires et non monotones où les conditions faisant intervenir plusieurs champs de l'entrée. L'algorithme 4 synthétise le fonctionnement de la combinaison faisant notamment intervenir une fonction d'ordonnancement.

Dans l'algorithme,  $T$  est l'ensemble des cas de test et  $Q$  la file de priorité gérée par la fonction de *fitness*. Cette fonction favorise les graines ayant découvert de nouveaux noeuds, défavorise celles ayant découvert de nouveaux chemins et jette celles n'ayant pas amélioré la couverture. Un avantage de combiner les deux techniques est que le test concolique *grey-box* ne peut pas augmenter la taille de la graine alors que le fuzzing *grey-box* lui peut. Dans l'algorithme  $R_R$ ,  $etR_G$  sont les ressources en nombre d'exécutions du programme (noté  $N_{exec}$ ) autorisées pour chacune des deux techniques. Lorsque les ressources d'une analyse sont épuisées, `Eclipser` interchange l'analyse. Il évalue aussi dynamiquement l'efficacité d'une analyse avec l'équation  $f = N_{path}/N_{exec}$  où  $N_{exec}$  est le nombre de cas de test ayant été exécuté et peut donc moduler les ressources en fonction. Cette technique répond directement à la question de recherche RQ\_COMB2.

**Algorithm 4** Algorithme Eclipser

---

```

1: function Eclipser( $p$ , seeds,  $t$ )
2:    $Q \leftarrow \text{InitQueue}(\text{seeds})$ 
3:    $T \leftarrow \emptyset$ 
4:   while  $\text{getTime}() < t$  do
5:      $R_G, R_R \leftarrow \text{Schedule}()$ 
6:      $Q, T \leftarrow \text{GreyConcolicLoop}(p, Q, T, R_G)$ 
7:      $Q, T \leftarrow \text{RandomFuzzLoop}(p, Q, T, R_R)$ 
8:   end while
9:   return  $T$ 
10: end function
```

---

### 13.6.5 Implémentation

Le projet s'appuie sur QEMU [26] pour l'instrumentation dynamique et le fuzzing pour lequel 800 lignes de code C ont été rajoutées. Le test concolique *grey-box* est implémenté en 4.4K lignes de F#<sup>161</sup> qui s'inspire d'AFL notamment pour la mutation auquel est rajouté un algorithme de *greedy set-cover* [291] pour la sélection de graines.

#### Langage F#

F# est un langage fonctionnel dérivé de OCaml, créé par Microsoft. Il est conçu pour être facilement interopérable avec du .NET et présente une syntaxe plus simple que OCaml. Il se différencie aussi par la simplicité d'implémenter une GUI, de faire du multi-threading et ajoute un support natif des types .NET dont les string unicode, les entiers 64 bits, etc. De plus il présente de meilleures performances que les autres langages fonctionnels comme OCaml ou Haskell.

L'article annonce un support de ARMv7 qui ne peut cependant pas être publié en raison de problèmes de propriété intellectuelle. Celui-ci ne verra donc jamais le jour dans l'outil.

### 13.6.6 Résultats

Les résultats obtenus dans la publication sont convaincants, car effectués en comparaison de AFLfast [39], LAF-intel [219], Steelix [235], VUzzer [290] et KLEE [56] (Qsym ayant été publié après la soumission de l'article et Angora, Steelix n'étant pas open-source au moment de la rédaction). Tous ces outils ont été comparés à LAVA-M. Les résultats sont donnés dans le tableau 13.3 (issu de l'article). A noter que les résultats de Steelix ont eux aussi été récupéré de l'article idoine [235].

---

161. <https://fsharp.org/>

Programme	AFLfast	LAF-intel	VUzzer	Steelix	Eclipser
base64	0	40	17	43	<b>46</b>
md5sum	0	0	1	28	<b>55</b>
uniq	0	26	27	7	<b>29</b>
who	0	3	50	194	<b>1135</b>
<b>Total</b>	<b>0</b>	<b>69</b>	<b>95</b>	<b>272</b>	<b>1265</b>

TABLE 13.3 – Résultats comparatifs LAVA-M Eclipser

Les résultats sont retentissants en faveur de **Eclipser** qui trouve presque 3x plus de bugs que tous les autres outils réunis. Il trouve le meilleur nombre de bugs pour chacun des binaires. Pour **base64** il en trouve d'ailleurs 46 alors que les auteurs de LAVA-M en référence uniquement 44.

Pour les restes des benchmarks, **Eclipser** obtient une amélioration de 8.57 % sur les **coreutils** par rapport à **KLEE**. Il a aussi permis de trouver 40 bugs uniques parmi 20 programmes issus de Debian 9.1 desquels 8 CVEs ont été créées.

#### Note

**Eclipser** a été intégré à **DeepState** [144] une interface de génération de tests unifiée pour effectuer aussi bien du fuzzing que du DSE. Leurs résultats montre qu'il obtient souvent de meilleurs résultats que **libfuzzer** voir parfois qu'AFL<sup>a</sup>.

a. <https://github.com/trailofbits/deepstate#fuzzing-with-eclipser>

### 13.6.7 Conclusion

L'approche guidée par la recherche *search-based* semble être une solution très porteuse a vu des résultats obtenus. Elle capitalise sur les résultats et les avancées effectuées par **Angora** et **Qsym** et pousse l'idée de contraintes approximées au point de ne plus avoir à les résoudre par SMT. C'est sans doute l'approche répondant au mieux à la problématique de recherche **RQ\_COMB1** questionnant la possibilité de faire passer le DSE à l'échelle (même si ça n'en est plus *stricto sensu*).

Aucune mention n'y ait fait dans la publication, mais cette approche revient à introduire un solveur de contraintes au sens **CP** du terme dans l'exécution du programme (voir Annexe D différence CP et SMT) basé sur un domaine d'intervalle. D'une certaine manière ils "redécouvrent" un peu la résolution de contraintes.

Le code, bien que performant, est écrit en F# et nécessite donc le runtime **mono** pour faire fonctionner le SDK .NET sur Linux. Le langage exotique utilisé et la nécessité de dépendre du SDK .NET<sup>162</sup> rendrait certainement la combinaison de cet outil avec un

162. <https://dotnet.microsoft.com>

autre assez difficile. Dans le cadre de PASTIS la combinaison pourrait être utilisée telle qu'elle est, mais nécessiterait d'ajouter le support pour ARM (*ce qui paraît incertain*).

## Chapitre 14

---

# Évaluation des outils

---

L'évaluation des combinaisons d'outils n'est pas aisée, car les approches des combinaisons se basent sur des principes hétérogènes. Il est difficile de créer des tests sans favoriser l'une ou l'autre des approches. Néanmoins, la suite de tests utilisée semble suffisamment variée pour limiter ce biais.

Parmi, les ressources étudiés, seul `Driller` et `cyberdyne` n'ont pas pu être testé. Le premier fonctionne exclusivement sur les binaires Linux DECREE et aucune des solutions proposées pour les porter vers des binaires x86 64 bits n'étaient utilisable en pratique dans le temps imparti à l'étude de la ressource. De même, bien que certains composants de `cyberdyne` soient maintenant open-source comme `GRR` ou `Manticore` (successeur de `PySymEmu`), le mixer, le minset et la machinerie effectuant l'interconnexion ne sont pas disponibles. De plus `GRR` aussi ne fonctionne que sur les binaires DECREE.

La dépendance de `Qsym` à `Pin` le rend extrêmement pénible à compiler. Ce dernier est notoirement connu pour être très contraignant sur les versions de la chaîne de compilation GCC et du noyau Linux. En raison de difficultés de compilation, `Qsym` a uniquement été testé sur le docker fourni par les développeurs, à savoir Ubuntu 14.04 et 16.04. Le faire fonctionner sous d'autres systèmes nécessiterait un effort non négligeable.

Dernière difficulté, aucun de ces outils ne supporte l'architecture ARM (sauf `Eclipsor` mais elle n'est pas libre). Par conséquent, seuls les benchmarks sur x86 ont pu être effectués.

### 14.1 Bombes logiques : Évaluation & Analyse des résultats

L'approche combinée permet théoriquement d'obtenir de meilleurs résultats que l'union du DSE et du fuzzing l'un tirant parti de l'autre. Les résultats de `Qsym`, `Angora` et `Eclipsor` aux tests atomiques est présenté sur le tableau 14.1 et permet d'avoir une idée des faiblesses des différentes approches de combinaisons (respectivement du DSE

sous-constraint, de la propagation de teinte pure et une approche *search-based* inspirée du DSE).

### 14.1.1 Résultat de UT\_1

Sans surprise la plupart des tests de cette unité sont résolus. L'exception est **CE\_2 Propagation**, qui demande à l'outil d'être capable de propager de la teinte à travers des écritures puis des lectures dans un fichier. Plus surprenant **stack\_cp\_11** échoue alors que la fonction de test est celle donnée en 25.

---

**Listing 25** Fonction de test de **CIId\_9 stack\_cp\_11** suite Kocwork du NIST (SARD)

---

```
1 int __attribute__((optnone)) entry(const char* s) {
2     int symvar = s[0] - 48;
3     int j;
4     __asm__ __volatile__("push %0" :: "m"(symvar));
5     __asm__ __volatile__("pop %0" :: "m"(j));
6     if(j == 7){
7         return TRIGGERED;
8     } else{
9         return NOT_TRIGGERED;
10    }
11 }
```

---

Même une gestion triviale de la mémoire devrait être en mesure de le résoudre à moins que l'instrumentation ajoutée niveau source désactive toute propagation de teinte sur l'assembleur inline. **Qsym** échoue aussi sur ce test alors qu'il fonctionne au niveau binaire. La résolution de **stackarray\_sm\_rw\_11** est particulièrement inopinée puisqu'elle implique de résoudre une écriture symbolique puis une lecture au même offset. L'explication la plus probable est la brutalité du fuzzing qui a fini par aligner les deux pointeurs (*évoluant quand même dans un espace restreint*).

### 14.1.2 Résultats de UT\_2

Dans cette unité l'échec de **Qsym** sur **malloc** et **memset** est particulièrement surprenant puisque **malloc\_1** se contente de tester que le pointeur retourné n'est pas nul. Il ne gère pas du tout les appels système (*qui requiert des callbacks spécifiques dans Pin*). Le test **syscall\_alarm** montre que la gestion des signaux est un point faible, quel que soit l'outil. Enfin, les tests **cpuid\_1** et **cpuid\_2** ne peuvent pas être résolus par ces approches très concrètes. Pour les résoudre, il faut avoir un CPU particulier ou être en mesure de symboliser les effets de bords de l'instruction assembleur. À noter le sans-faute de **Eclipser** sur la gestion des strings et les flottants.

#### 14.1. Bombes logiques : Évaluation & Analyse des résultats

	Id	Nom	Qsym 0.1		Angora 1.0.0		Eclipser 0.1	
			x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_1	CE_1	CId_1	array_sample1	✓	-	✓	-	-
		CId_2	array_sample2	✓	-	✓	-	✓
		CId_3	array_sample3	✓	-	✓	-	✓
		CId_4	array_sample4	✓	-	✓	-	✓
		CId_5	array_sample5	✓	-	✓	-	✓
		CId_6	array_sample6	✓	-	✓	-	✓
	CE_2	CId_7	file_cp_11	✓	-	✗	-	✓
		CId_8	file_posix_cp_11	✓	-	✗	-	✓
		CId_9	stack_cp_11	✓	-	✗	-	✓
	CE_3	CId_10	stackarray_sm_11	✓	-	✓	-	✓
		CId_11	stackarray_sm_12	✓	-	✓	-	✓
		CId_12	stackarray_sm_ln	✗	-	✓	-	✓
		CId_13	pointers_sj_11	✓	-	✓	-	✓
	CE_4	CId_14	stackarray_sm_store_11	✓	-	✓	-	✓
		CId_15	stackarray_sm_store_12	✓	-	✓	-	✓
	CE_5	CId_16	stackarray_sm_rw_11	✓	-	✓	-	✓
UT_2	CE_1	CId_17	atoi_ef_12	✓	-	✗	-	✓
		CId_18	pow_ef_12	✓	-	✓	-	✓
		CId_19	printint_int_11	✓	-	✓	-	✓
		CId_20	rand_ef_12	✓	-	✗	-	✓
		CId_21	file_csv	✓	-	✗	-	✗
		CId_22	file_posix_csv	✓	-	✗	-	✗
		CId_23	pid_csv	✓	-	✓	-	✓
		CId_24	malloc_1	✗	-	✓	-	✓
		CId_25	memset_1	✗	-	✓	-	✓
		CId_26	syscall_alarm	✗	-	✗	-	✗
	CE_2	CId_27	syscalls_open_close	✓	-	✓	-	✓
		CId_28	syscalls_read_write	✓	-	✓	-	✓
		CId_29	syscall_time	✗	-	✗	-	✓
	CE_3	CId_30	pmccntr	✗	-	✓	-	✓
		CId_31	cpuid_1	✗	-	✗	-	✗
		CId_32	cpuid_2	✗	-	✗	-	✗
	CE_4	CId_33	string_sample1	✗	-	✗	-	✓
		CId_34	string_sample2	✓	-	✗	-	✓
		CId_35	string_sample3	✓	-	✓	-	✓
		CId_36	string_sample4	✓	-	✗	-	✓
		CId_37	propagation_sample3	✗	-	✗	-	✓
	CE_5	CId_38	simple_float1	✓	-	✓	-	✓
		CId_39	atof_ef_12	✓	-	✗	-	✓
		CId_41	sin_ef_12	✓	-	✗	-	✓
		CId_42	ln_ef_12	✓	-	✓	-	✓
		CId_43	5n+1_lo_11	✓	-	✓	-	✓
UT_3	CE_1	CId_44	collaz_lo_11	✓	-	✓	-	✓
		CId_45	loop_sample1	✓	-	✓	-	✗
		CId_46	loop_sample2	✓	-	✓	-	✓
		CId_47	loop_sample3	✗	-	✗	-	✗
		CId_48	df2cf_cp_11	✓	-	✓	-	✓
	CE_2	CId_49	sample_path1	✗	-	✗	-	✗
		CId_50	sample_path2	✗	-	✗	-	✗
		CId_51	sample_path3	✗	-	✗	-	✗
		CId_52	sample_path4	✗	-	✗	-	✗
		CId_53	sample_path5	✓	-	✗	-	✓
		CId_54	sample_path6	✓	-	✓	-	✓
UT_4	CE_1	CId_55	bof_sample1	✓	-	✗	-	✓
		CId_56	bof_sample2	✗	-	✗	-	✓
		CId_57	stack_bo_11	✓	-	✗	-	✓
	CE_2	CId_58	iof_sample1	✗	-	✓	-	✓
		CId_59	iof_sample2	✗	-	✓	-	✓
		CId_60	integer_overflow_1	✗	-	✓	-	✓
	CE_3	CId_61	off_by_one_1	✗	-	✗	-	✓
		CId_62	off_by_one_2	✓	-	✗	-	✓
		CId_63	off_by_one_3	✓	-	✗	-	✓
	CE_4	CId_64	stackarray_sm_12_off_by_one	✗	-	✓	-	✓
		CId_65	use_after_free_1	✓	-	✓	-	✓
		CId_66	use_after_free_2	✓	-	✓	-	✓
	CE_5	CId_67	format_string_1	✓	-	✓	-	✓
		CId_68	format_string_2	✓	-	✓	-	✓
		CId_69	format_string_3	✓	-	✗	-	✓
	CE_6	CId_70	read_anywhere	✗	-	✓	-	✗
	CE_6	CId_70	stackoutofbound_sm_12	✗	-	✓	-	✓

TABLE 14.1 – Résultats détaillés des benchmarks de fuzzing sur les tests atomiques

### 14.1.3 Résultats de UT\_3

Le test `CId_45 : loop_sample_1` calcule un “serial” à partir de l’entrée. Un test avec un plus gros timeout a montré que `Eclipser` l’aurait résolu ( 10 mins). Le problème ici est donc le timeout. De surcroît, il est très déconcertant que la majorité des tests `sample_path` ne soient résolus par aucun outil tant ils sont simples.

### 14.1.4 Résultats du UT\_4

Les débordements d’entiers sont relativement mal gérés par `Qsym`. Ce phénomène et difficilement explicable, car les tests sont facilement solubles par exécution symbolique. Malheureusement `Angora` peine à détecter les différents types de débordements de buffers, qu’ils soient d’un ou plusieurs octets. À l’inverse, il est le seul à arriver à résoudre les `read anywhere`.

## 14.2 LAVA-M : Évaluation & Analyse des résultats

Les résultats obtenus sur LAVA-M sont extraordinairement bons puisqu’ils trouvent pour la quasi-totalité plus de bugs insérés qu’originellement prévu par les concepteurs de LAVA. Le tableau 14.2 synthétise les résultats obtenus (*encore une fois sur x86*).

	CE	Nom	Qsym 0.1		Angora 1.0.0		Eclipser 0.1	
			x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_5	CE_1 : Nombre de bugs trouvés	base64	48/44	-	48/44	-	45/44 (46*)	-
		uniq	29/28	-	29/28 <sup>163</sup>	-	23/28 (29*)	-

\* :résultats de l’article de recherche

TABLE 14.2 – Résultats des benchmarks pour UT\_5 : Passage à l’échelle

Avec `Qsym` sur `uniq`, l’union des trois exécutions de 6h à permis de trouver les 29 bugs alors que pris indépendamment ils en trouvaient un de moins. Pour `Eclipser` il y a une légère disparité entre les résultats obtenus et ceux du papier qui en plus étaient exécutés 5h et non 6h. Il est probable qu’un paramètre soit légèrement mieux configuré dans leurs benchmarks.

Les benchmarks ne reflètent pas cette donnée, mais les trois outils trouvent la plupart des bugs avec une rapidité fulgurante. Le temps alloué de 6h n’est donc pas nécessaire puisque les outils atteignent très vite l’asymptote des bugs à trouver. Ensuite, ils se contentent de réactiver les mêmes bugs avec différentes entrées. À titre d’exemple, sur 45 bugs uniques dans `base64`, `Eclipser` en trouve en moyenne 1503 et 1901 pour `uniq`.

163. résultats de l’article (*à cause d’une erreur inconnue lors de l’exécution des tests*)

## 14.3 Synthèse & Conclusion des benchmarks

Les résultats obtenus sur LAVA avec les DSE étaient catastrophiques (cf. 8.3), tout comme les résultats d’AFL étaient aussi très décevants (cf. 11.2). Combinés ici, comme c’est le cas pour **Qsym**, les résultats sont pour le moins impressionnantes. **Qsym** améliore drastiquement les résultats de AFL utilisé seul. L’intérêt d’une combinaison est donc entièrement démontré.

En comparaison d’un fuzzing whitebox standard, ces trois approches tirent profit d’une approche relaxée des contraintes permettant un passage à l’échelle sensationnel. Parmi, les analyses utilisées sans combinaison, KLEE concurrence les combinaisons sur les tests atomiques et Honggfuzz les concurrences sur les deux aspects (cf. 11.3). Le tableau 14.3 donne un aperçu plus quantitatif des performances de chaque outil.

	Critères	Qsym 0.1		Angora 1.0.0		Eclipser 0.1	
		x86_64	ARM	x86_64	ARM	x86_64	ARM
UT_1	CE_1 : Tableau	6/6	-	6/6	-	6/6	-
	CE_2 : Propagation	3/3	-	0/3	-	3/3	-
	CE_3 : Lecture mémoire symbolique	3/4	-	4/4	-	4/4	-
	CE_4 : Ecriture mémoire symbolique	2/2	-	2/2	-	2/2	-
	CE_5 : Lecture/Ecriture mémoire symbolique	1/1	-	1/1	-	1/1	-
UT_2	CE_1 : Fonctions externes	7/9	-	5/9	-	7/9	-
	CE_2 : Appels système	2/4	-	2/4	-	3/4	-
	CE_3 : Instructions non-déterministes	0/3	-	1/3	-	1/3	-
	CE_4 : Gestion des strings	3/5	-	1/5	-	5/5	-
	CE_5 : Nombres flottants	4/4	-	2/4	-	4/4	-
UT_3	CE_1 : Boucles	4/5	-	4/5	-	3/5	-
	CE_2 : Chemins	3/7	-	2/7	-	3/7	-
UT_4	CE_1 : débordement de tampon	2/3	-	0/3	-	3/3	-
	CE_2 : débordement d’entiers	0/3	-	3/3	-	3/3	-
	CE_3 : Off-by-one	2/4	-	1/4	-	4/4	-
	CE_4 : Use-after-free	2/2	-	2/2	-	2/2	-
	CE_5 : Format string	3/3	-	2/3	-	3/3	-
	CE_6 : Accès mémoire invalide	0/2	-	2/2	-	1/2	-
<b>Total bombes logiques :</b>		<b>47/70</b>	-	<b>40/70</b>	-	<b>58/70</b>	-
UT_5	CE_1 : Nombre de bugs trouvés <code>base64</code>	48/44	-	48/44	-	45/44	-
	CE_1 : Nombre de bugs trouvés <code>uniq</code>	29/28	-	29/28	-	23/28	-
<b>Total :</b>		<b>124/142</b>	-	<b>117/142</b>	-	<b>126/142</b>	-

TABLE 14.3 – Résultats synthétiques des benchmarks des outils de combinaison

Les résultats obtenus sont à nuancer, car la méconnaissance des outils et des paramètres d’entrées rend certainement les résultats sous-optimaux. De plus, les tests atomiques non résolus de manière inexplicable sont probablement dus à la nature des binaires fournis. En effet, tous étaient compilés dynamiquement alors que la plupart des outils fonctionnent mieux sur des binaires statiques. Par exemple, AFL n’instrumente pas les bibliothèques (*library*) dynamiques donc un simple `atoi` peut mettre en échec le test.

Les résultats du tableau 14.3 sont particulièrement intéressants, car ils permettent de comparer **Eclipser** à **Qsym** et **Angora** ce que les auteurs de l’article n’avaient pas pu faire, car **Qsym** n’était pas encore publié et **Angora** pas encore open-source. Les résultats valident l’efficacité de cette approche qui dépasse les deux autres d’un point de vue purement quantitatif. Ces trois approches différentes sont donc à considérer dans le cadre du projet PASTIS.

## **Sixième partie**

### **Slicing**



## État de l'art : Slicing

---

### 15.1 Introduction

Le slicing (ou simplification syntaxique) est une technique permettant d'extraire un sous-ensemble d'un programme appelé slice à partir d'un critère (une ou plusieurs instructions), tout en conservant le comportement du programme initial vis-à-vis du critère<sup>164</sup>. Cette notion a été introduite par Weiser en 1984 [361] et désigne initialement un sous-programme exécutable. Depuis, la définition de slice a évolué et désigne aussi un sous-ensemble (non exécutable) d'instructions du programme. Cette technique, initialement appliquée à des programmes C, est en fait indépendante du langage de programmation [358, 359] (*moyennant quelques spécificités liées aux paradigmes du langage*).

Les trois principales caractéristiques d'un algorithme de slicing sont les suivantes :

**Type** le slice peut être statique ou dynamique et, dans ce cas, calculé à partir d'une entrée du programme ;

**Direction** avant ou arrière. À partir d'une cible, il suit les dépendances en avant ou en arrière ;

**Exécutabilité** définit si le slice est compilable et exécutable ou si c'est une clôture (*closure*) [350] ; dans le second cas, on parle aussi de slice « logique », car les contraintes pour que l'exécution reste dans le slice sont appliquées de manière logique.

Dans le contexte de PASTIS, le slicing se présente comme une analyse complémentaire au fuzzing et au DSE afin d'améliorer leur couverture de code et de les guider mécaniquement vers les cibles choisies. Les slicers statiques en arrière appliqués avant le DSE ou le fuzzing sont donc les plus à même de satisfaire les attentes de PASTIS.

Cet état de l'art s'appuie sur différentes sources dont le *survey* de Frank Tip publié en 1994 [343] et d'autres travaux plus récents [225, 369].

---

164. pour la suite, slice sera désigné comme un sous-ensemble du programme et donc comme un mot masculin (en opposition à une tranche la traduction littérale de slice)

## 15.2 Algorithmes

Le but de tous les algorithmes de slicing est de calculer le slice minimal tel que toutes les conditions et variables conservées dans le slice impactent effectivement le critère choisi. Or, Weiser a montré l'indécidabilité de ce problème [361]. Une définition plus précise d'un slice est donc la suivante :

**Définition 6.** *Un slice est une sur approximation conservative du slice minimal, et plus précisément, des instructions (statements) qui peuvent avoir une influence sur le critère de slice.*

Le critère de slice est habituellement défini par le tuple  $(s, V)$ , avec  $s$  le *statement* (définition) et  $V$  une ou plusieurs variables que l'on veut slicer. Un *statement* peut se définir par un 2-uplet  $(l, \text{decl})$  où  $l$  est le numéro de ligne (ou l'adresse) et `decl` une déclaration ou une instruction dans le langage cible.

**Slicing en avant.** Le slicing était initialement conçu pour fonctionner en arrière, mais il suffit d'itérer les dépendances en avant depuis le critère de slicing pour obtenir un « slice en avant ». Celui-ci ne représente alors plus ce qui impacte le critère, mais plutôt ce que le critère impacte. Ce type d'analyse est couramment appelé *analyse d'impact*. Les algorithmes de calcul de slice avant et arrière sont respectivement détaillés dans les sections 15.3 et 15.4.

**Dicing** et **chopping** sont deux algorithmes dérivés du slicing. Le *dicing* [362] crée un *dice* qui correspond à la différence entre un slice d'une variable invalide et le slice d'une variable valide. Cela permet de localiser les instructions qui affectent la variable invalide sans impacter la variable valide. Pour du debugging, cela permet de restreindre les lignes de code (instructions) à inspecter. Le *chopping* [190, 295] résout le problème de savoir « *comment une variable en affecte une autre* ». Cette technique combine slicing en avant à partir d'un critère *source*, et slicing en arrière à partir d'un critère *cible*. Le slice résultant est donc l'intersection des deux slices.

## 15.3 Slicing statique

Plusieurs approches de calcul des slices statiques ont été proposées. Les premières furent des approches itératives sur les équations *data-flow*, autrement dit, le lien entre une affectation de variable et son expression. D'autres furent développées en utilisant les relations sur le flot d'information du programme [27]. Néanmoins, les algorithmes les plus utilisés se basent sur l'interprétation abstraite (*Abstract Interpretation (AI)*) ou sur des modèles graphiques, représentations des dépendances sous forme de graphe appelé *Program Dependency Graph (PDG)* introduites en 1984 par Ottenstein [270]. Toute la difficulté est de calculer ce PDG – présenté dans la section ci-dessous 15.3.2. Un *PDG* se calcule au niveau fonction (procédure) ; pour obtenir un slice interprocédural, il faut étendre ce graphe sous la forme d'un *System Dependency Graph (SDG)* dont la notion a été introduite par Thomas Reps en 1990 [177] (cf. Section 15.3.2).

### 15.3.1 Slicing à base d'interprétation abstraite

Plusieurs algorithmes de calcul de slicing s'appuient sur l'interprétation abstraite comme base d'analyse [6, 249]. Des travaux effectués plus récemment [120] proposent l'utilisation d'une interprétation abstraite pour extraire des slices liés à la détection de use-after-free. L'intuition est d'identifier par AI les trois artefacts clés d'un use-after-free, à savoir, une allocation `malloc`, une désallocation `free` et un usage impropre `use`. Le domaine abstrait utilisé est un triplet représentant l'état de ces trois paramètres pour chaque variable et plus particulièrement les pointeurs. Lorsque ces trois artefacts sont trouvés, un slice les englobant est créé. La couverture et la validation de la vulnérabilité sont ensuite effectuées dans ce slice. Plus de détails sur la combinaison avec la couverture par DSE sont donnés Section 15.6.2.

Quelque soit la méthode d'extraction du slice, toute l'intelligence de l'algorithme est déportée dans l'AI. L'efficacité et la précision du slice sont entièrement dépendantes de celles de l'analyse par interprétation abstraite.

### 15.3.2 Slicing à base de dépendance de programme

Les algorithmes de slicing se basant sur le PDG ramènent le problème du slicing à un problème d'atteignabilité sur ce graphe [293]. Le calcul de ce graphe s'appuie sur différents graphes intermédiaires, tel que représenté sur le schéma 15.1.

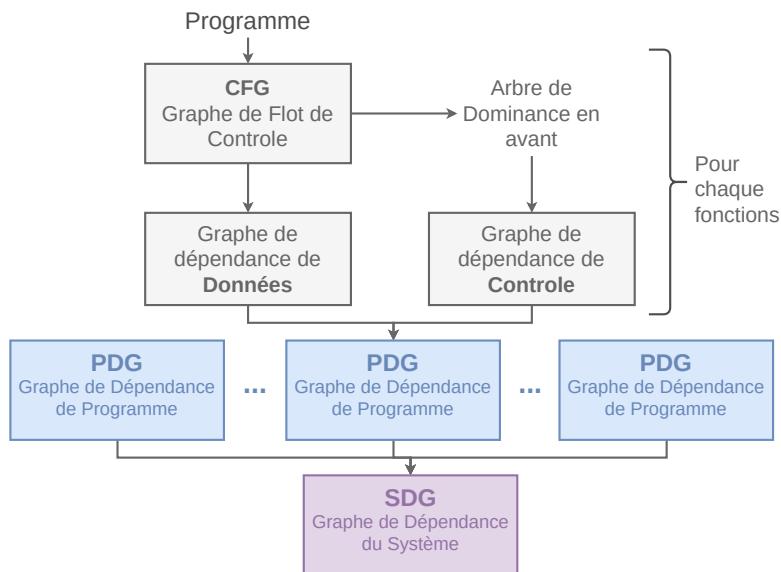


FIGURE 15.1 – Étapes de génération d'un SDG.

La représentation de base est le CFG d'une fonction à partir duquel il est possible de dériver le graphe de dépendances de contrôle et celui de dépendances de données. Ces deux graphes peuvent ensuite être fusionnés pour obtenir le PDG. Les PDG obtenus

**Listing 26** Code d'illustration du slicing (src. [185])

```

1 void main() {
2     int i = 1;
3     int sum = 0;
4     while (i<11) {
5         sum = add(sum, i);
6         i = add(i, 1);
7     }
8     printf("sum = %d\n", sum);
9     printf("i = %d\n", i);
10 }
11
12 static int add(int a, int b){
13     return (a+b);
14 }
```

---

pour chaque fonction peuvent à leur tour être fusionnés pour obtenir le SDG. Différentes structures de données comme les sauts indirects et les sauts inconditionnels viennent entraver le calcul de ces différents graphes qui est loin d'être trivial. Pour la suite, le Listing 26 illustre un programme simple avec une boucle effectuant un appel à une seconde fonction.

## Dépendance de données

**Dépendances de données.** Le calcul s'effectue sur le CFG et requiert en général une analyse de *def-use* et une analyse de pointeurs (pour être suffisamment précis). Cette première calcule les ensembles  $DEF(i)$  et  $USE(i)$  qui représentent respectivement la localisation  $i$ , l'ensemble des variables définies et l'ensemble des variables référencées. On dit qu'un noeud  $j$  est dépendant de  $i$  s'il existe une variable  $x$  telle que  $x \in DEF(i)$ ,  $x \in USE(j)$  et s'il existe un chemin entre  $i$  et  $j$  ne contenant aucune redéfinition de  $x$ . La génération du graphe de dépendance de données soulève deux principaux problèmes.

**Structures et tableaux.** Le premier est celui de la gestion des tableaux, qu'il est soit possible de les considérer comme une seule variable soit de les décomposer en cellules indépendantes. Dans le second cas, le tableau requiert une analyse de dépendance beaucoup plus approfondie qui implique de raisonner sur les pointeurs.

**Aliasing de pointeur.** Le deuxième problème, le plus important, est la gestion des pointeurs et du potentiel *aliasing*, dont le calcul est non décidable [221]. Ce problème, semblable à RQ\_DSE1, sera noté RQ\_SLC1 dans le contexte du slicing statique. La précision du slice est directement dépendante de la précision de l'analyse d'alias. De plus, le

slicing en présence d'aliasing requiert une généralisation de la notion de dépendance pour prendre en compte ces alias. Ce problème reste un problème de recherche ouvert [173] et aucune solution optimale n'a été trouvée. Les travaux [178] proposent une alternative à la notion de dépendance par définition/utilisation (*def-use*) de variables basées sur la définition/utilisation d'emplacements mémoire abstraits. Cela introduit un niveau de complexité supplémentaire puisque la dépendance entre les "def" et les "use" de mémoire abstraites ne sont plus binaires. Trois cas peuvent se produire :

**intersection complète** la définition est un surensemble de l'utilisation.

**intersection potentielle** (*maybe*) , il est impossible de déterminer si la définition d'une mémoire  $e_1$  intersecte avec l'utilisation de la mémoire  $e_2$  (e.g.,  $e_1 = a[i]$  et  $e_2 = a[j]$  où  $i$  et  $j$  sont inconnus).

**intersection partielle** la définition est un sous-ensemble de l'utilisation (e.g.,  $e_1$  définit  $a[5]$  et  $e_2$  utilise  $a[i]$ ).

## Dépendance de contrôle

Les dépendances de contrôle s'expriment en termes de post-dominance en théorie des graphes sous la forme d'un graphe acyclique **Directed Acyclic Graph (DAG)**. Dans le CFG, un noeud  $i$  est "post-dominé" par  $j$  si tous les chemins de  $i$  au noeud de sortie passent par  $j$ . Le noeud  $j$  est alors dépendant (de contrôle) sur  $i$  [121].

## Dépendance de programme PDG

Une fois le graphe de dépendance de contrôle et le graphe de dépendance de données calculés, ceux-ci peuvent être fusionnés pour former le **PDG** illustré en Figure 15.2. Le **PDG** est un **DAG** dont les noeuds représentent les déclarations du programme (*statement*) et les arêtes les dépendances aussi bien de données que de contrôles. Les arêtes sont étiquetées en fonction de la dépendance qu'elles représentent. Selon cette définition, un slice est l'ensemble des noeuds à partir desquels le noeud d'intérêt (critère de slice) peut être atteint. Sur la figure 15.2, les flèches bleues représentent les dépendances de contrôle (issues de l'arbre de post-dominance) et les flèches vertes les dépendances de données.

## Dépendance de système

Le calcul du **SDG** tel que définit par Reps [177] est appelé HRB (Horwitz-Reps-Binkley) et implique les trois étapes suivantes (un algorithme détaillé est donné en Section 16.1.2) :

- agréger les **PDG** de chaque procédure dans le même graphe **SDG** ;
- compléter le **SDG** avec des noeuds de résumé (*summary edges*) représentant les dépendances transitives entre les paramètres de la fonction et les paramètres des fonctions qui l'appellent ;

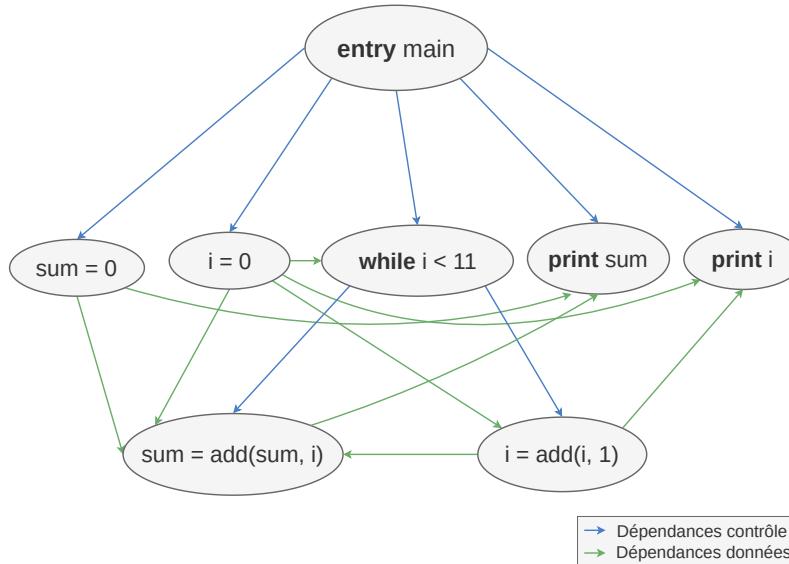


FIGURE 15.2 – PDG pour la fonction `main` de l'exemple du Listing 26.

- calculer le slice avec un parcours du [SDG](#) en deux passes. La première suit les dépendances intraprocedurales et la deuxième descend dans les nœuds d'appels pour calculer le slice de manière interprocédurale.

Lors de la construction du [SDG](#), pour chaque instruction `call` un nœud est créé ainsi que des nœuds d'entrées et de sorties (*actual-in*, *actual-out*) qui représentent la copie des paramètres d'entrées/sorties entre les fonctions (*summary edges*). La complexité de l'algorithme HRB est cruciale et plusieurs optimisations ont été proposées pour améliorer cet algorithme et en particulier le calcul des *summary edges* [125, 294]. Les spécificités et les difficultés liées au calcul d'un [SDG](#) à partir d'un binaire ont aussi été étudiées [210].

Le schéma 15.3 montre le [SDG](#) obtenu après application de l'algorithme sur les différents [PDG](#). Dans ce graphe les `call` sont représentés par un nœud auquel est associé un ensemble de nœuds représentant les paramètres immutables appelés *actual-in* (en orange) et les paramètres pouvant être modifiés par l'appel de la fonction appelée *actual-out* (en violet). Les variables globales utilisées par une fonction doivent aussi être représentées avec ce mécanisme. Ensuite les différentes dépendances de données et de contrôle sont propagées entre les deux fonctions (arêtes en pointillées). Ensuite, les slices et les “chops” (slices bi-directionnels) peuvent être calculés en parcourant le graphe.

**Problème du contexte.** L'aspect interprocédural de ce graphe pose un problème de taille lors du parcours. En effet, lors d'un `call`, il oblige l'algorithme de parcours à ressortir sur l'adresse de retour (*returnsite*) lorsqu'un `return` est rencontré. Sinon l'algorithme peut ressortir sur l'adresse de retour d'un autre appel ce qui fait peu de sens (*le slice sans être faux serait extrêmement imprécis*). Il faut être sensible au contexte d'appel (*context-sensitivity*). Ce problème de recherche sera dénoté [RQ\\_SLC2](#) par la suite. Toutes les solutions nécessitent de connaître le chemin déjà emprunté. La solution la plus

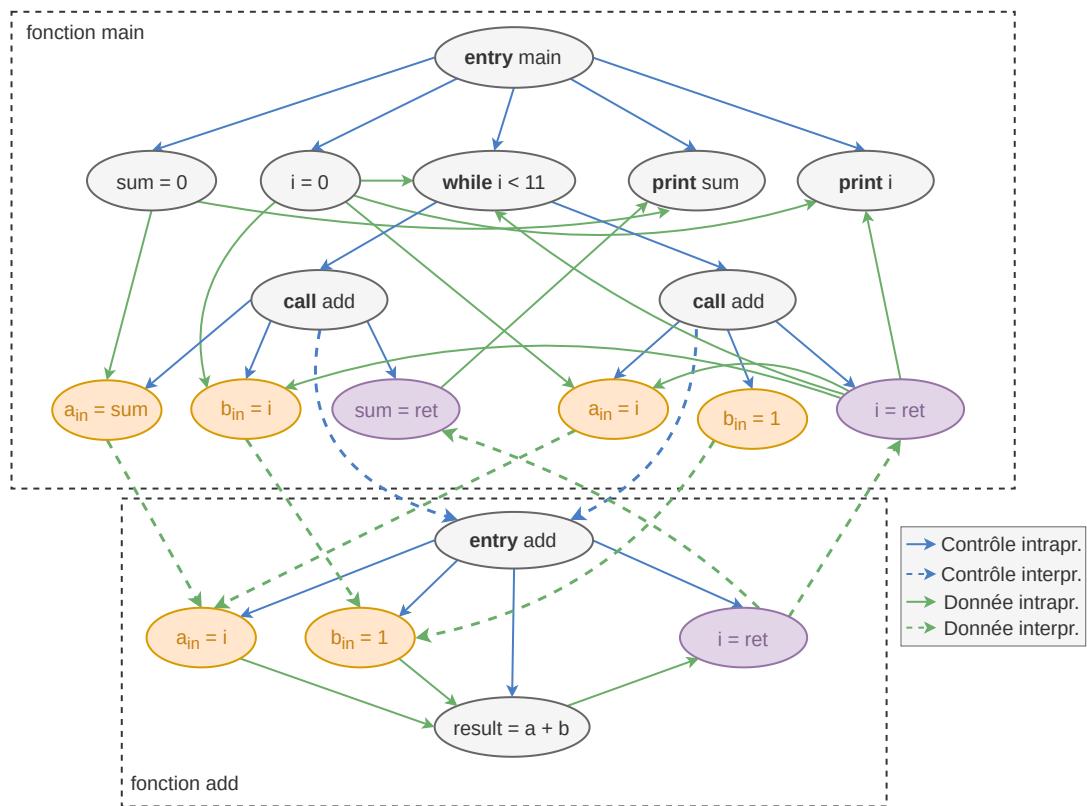


FIGURE 15.3 – SDG de l'exemple du Listing 26.

usité et de conserver une pile d'appels [168, 322]. Toutefois, certaines fonctions ne retournent pas à l'appelant (tail calls, setjmp-longjmp), c'est pourquoi des extensions ont été proposées pour adresser les CFG déstructurés via l'idée d'[Augmented Control Flow Graph \(ACFG\)](#) [322]. Les ACFG, sont des CFG auxquels sont ajoutés des arêtes sur les dépendances de contrôle interprocédurales.

## 15.4 Slicing dynamique

Le slicing dynamique introduit en 1988 [214] se différencie par le fait qu'il n'est valide que pour une entrée donnée, alors qu'un slice statique est valide pour toutes les entrées. L'avantage est qu'il est beaucoup plus précis. Un slice dynamique différencie aussi les occurrences d'une même instruction. Un critère de slicing se définit par le n-uplet  $(tcs, occ, s, V)$  où  $tcs$  est l'entrée du programme et  $occ$  l'occurrence du *statement* (instruction) situé au *statement*  $s$ . À noter que cette définition implique une taille fixe sur l'entrée alors que le slicing statique ne fait aucune hypothèse sur sa longueur.

Plusieurs algorithmes ont été proposés, l'un d'eux, se base sur la notion de *dynamic dependence graph* (DDG) où chaque *statement* (déclaration) de la trace d'exécution est un noeud du graphe [5]. Une représentation réduite *Reduced-DDG* (RDDG) peut être calculée pour réduire la taille sans perdre de précision. Néanmoins, dans le pire cas, la taille du slice reste exponentielle par rapport au nombre de *statement*. D'autres représentations ont ensuite été proposées pour améliorer la compacité du slice ou pour améliorer le temps pour le calculer [147, 259]. Ces algorithmes sont conçus pour du slicing en arrière. En avant, l'algorithme habituellement utilisé [29] suit les définitions et les utilisations de variables (*def-use chains*) pour calculer le slice. Dans ce cas aucun graphe de dépendance n'est nécessaire.

### Différence avec analyse de teinte

Alors que le slicing est essentiellement syntaxique, l'analyse de teinte se distingue avant tout par son caractère *data-flow* et par l'aspect sémantique du traitement des instructions. Les différences sont décrites en annexe F.

## 15.5 Autres approches de slicing

En plus des démarches dominantes présentées ci-dessus, plusieurs approches plus séculières ont été proposées dans la littérature :

- Slicing “conditionné” [62] : il calcule un slice satisfaisant un critère à partir d'un ensemble de chemins d'exécution générés statiquement, eux-mêmes générés à partir d'un ensemble d'états initiaux. Les chemins ne satisfaisant pas ces états initiaux par exécution symbolique sont éliminés.

- Slicing conditionné basé sur les assertions d'un programme [24] ou sur les pre/post condition d'un contrat de fonction [86, 167, 229] ; ce sont des approches utilisant les artefacts et des annotations du programme comme critère de slice ; les travaux [167] proposent notamment un framework uniifié de slicing avant et arrière conditionné, poussé à l'automatisation complète [126] grâce à l'exécution symbolique (en Prolog) et la preuve de programme (en Isabelle<sup>165</sup>).
- Slicing “quasi-statique” [350] : il propose une approche hybride entre statique et dynamique en basant le slice non pas sur une, mais sur plusieurs exécutions (*explorant un sous-ensemble des entrées du programme*).
- Slicing amorphe (*amorphous slicing* [166]) : il étend la suppression des instructions hors du slice à d'autres composants et permet ainsi d'obtenir des slices plus compacts (améliore potentiellement le calcul de SDG cf. 15.3.2).
- Slicing d'union (*union-slicing* [28]) : c'est une approche dynamique visant à unifier plusieurs slices dynamiques générés avec des entrées différentes. L'approximation générée est donc beaucoup plus compacte qu'un slice statique.
- Slicing basé sur l'observation ((*observation-based*) [32]) : il s'appuie sur l'idée qu'il est possible de tester la dépendance d'un critère à une instruction en la supprimant puis en vérifiant que le programme se compile et produit toujours le même comportement. Si c'est le cas, l'instruction peut effectivement être exclue du slice.
- Slicing hybride [156] propose de réduire l'imprécision d'un slice statique et la spécificité d'un slice dynamique (une seule exécution) en combinant les deux. Par conséquent, l'idée est d'enrichir le slice statique avec des informations dynamiques.

## 15.6 Applications du slicing

Comme vu précédemment, un slice se calcule de plusieurs manières différentes et chaque cadre d'application nécessite un type de slice spécifique. Le type de slice à utiliser dépend donc entièrement de l'usage qui en est fait et du problème qu'il résout. Les cas d'application historiques sont le debugging, le test logiciel, la compréhension ou la maintenance de programme ou encore la parallélisation. L'usage du slicing “post crash” représente la majorité des cas d'usage. L'approche visant à effectuer le slicing comme pré analyse de fuzzing ou de DSE est moins répandue.

Le slicing trouve maintenant d'autres applications comme le Worst-Case-Execution-Time (WCET) sur des systèmes Real Time Operating System (RTOS) avec BEST [247] et plus communément dans le domaine du test et de la vérification logicielle comme Symbiotic [323], SANTE [68], Unravel [285] pour de la vérification de programmes. Des travaux visent même à faire de la preuve d'algorithme de slice pour obtenir de meilleures garanties formelles sur les analyses subséquentes [225].

En sécurité, il est principalement utilisé pour faire de l'analyse de cause racine (*root-cause analysis*) dans le cadre de recherche de vulnérabilités. Il sert donc plus pour assister l'analyste à la rétro-ingénierie pour améliorer la couverture de programme. Dans le

165. <https://isabelle.in.tum.de>

contexte de *root-cause analysis*, c'est plutôt un slice dynamique sur une trace d'exécution qui est utilisé. L'outil `moflow` [193] utilise `Pin` et le langage intermédiaire `BIL` de `BAP` [49] pour le calcul du slice.

### 15.6.1 Fuzzing

Peu de techniques de fuzzing emploient du slicing, les *fitness-function* servant naturellement à guider la recherche dans les zones prometteuses du programme. À ce titre, l'outil `MutaGen` [201] est particulièrement singulier puisqu'il propose d'utiliser du slicing non pas sur le programme cible, mais sur le programme générant les entrées pour le programme cible (générateur de pdf, mp3, tiff etc). Le slicing sert donc à la génération d'un corpus d'entrées envoyé ensuite au fuzzer, le but étant de générer des entrées produisant une grosse couverture du programme cible. L'avantage est que par défaut le programme de génération de fichiers produit des fichiers valides (*le fuzzer ne perd pas de temps à générer une structure valide par mutation*). Dans ce contexte, `MutaGen` utilise du slicing dynamique. Il génère d'abord (avec `Pin`) un *dyDDG dynamic data dependence graph*. Sur ce graphe, à chaque écriture d'un octet du fichier de sortie, un slice est réalisé pour conserver les instructions ayant contribué à la génération de l'octet. Ce processus est répété pour chaque octet, produisant ainsi un *slice dynamique cumulatif*. Sur ce slice sont appliquées des mutations (appliquées directement sur du `VEX` ensuite “JITé” par `Valgrind`) permettant de générer des fichiers mutés.

### 15.6.2 Exécution symbolique

L'usage combiné de slicing et d'exécution symbolique a trouvé plusieurs cas d'applications. Les travaux [191] utilisent l'exécution symbolique pour aider le slicing en arrière en coupant les suffixes de chemins qui se trouvent être infaisables. Cela se rapproche du `BB-DSE` [22] présenté en Section 6.7.3. L'idée est de réduire la taille du slice pour mieux passer à l'échelle.

L'outil `Symbiotic` [323] combine slicing, instrumentation et exécution symbolique pour faire de la vérification logicielle. Cet outil, et plus particulière son slicer `dg`, sont décrits plus en détail en Section 16.2.

Les travaux très récents autour de `Chopper` [345] proposent une approche de *Chopped Symbolic Execution* dans laquelle l'exécuteur symbolique évolue dans un “chop” et tout ce qui est en dehors est exécuté de manière complètement concrète. Cette méthode judicieuse s'appuie sur `KLEE` [56] pour l'exécution symbolique, sur `dg` (ci-dessus) pour le slicing et sur `SVF` [334] pour une analyse de dépendance et une analyse de pointeurs précise<sup>166</sup>.

**Slicing et use-after-free.** Ces travaux [120] implémentés dans `GUEB`<sup>167</sup> s'appuient sur l'interprétation abstraite pour le calcul du slice (cf. 15.3.1). Le slice sert à englober la

---

166. <http://svf-tools.github.io/SVF>

167. <https://github.com/montyly/gueb>

chaine `malloc-free-use`, à partir duquel le DSE effectue la couverture (en l'occurrence Binsec [99]) pour valider la présence d'un use-after-free. Le slice a la particularité d'être pondéré par la distance aux différents artefacts. Le but est de guider le DSE pour qu'il passe sur les trois artefacts dans l'ordre, sans quoi la vulnérabilité ne peut pas être testée. Dans cette application, le slice est appliqué "logiquement" à l'algorithme de couverture de DSE qui est contraint de rester à l'intérieur. L'implémentation a été abandonnée peu après la rédaction du papier, elle est donc inutilisable en pratique, mais l'idée reste néanmoins la plus proche de l'usage qui pourrait être fait du slicing dans le projet PASTIS.



## Chapitre 16

---

### Analyse détaillée des outils

---

Comme vu précédemment, l'utilisation voulue du slicing dans PASTIS est celle d'une pré-analyse auxiliaire au fuzzing et au DSE afin de faciliter la couverture de code et de les guider plus efficacement vers les objectifs de test. Le besoin se tourne donc plus vers des slicers statiques au niveau source. Or, les outils fournissant du slicing comme [Triton](#) [304] ou [angr](#) [320], sont plutôt orientés "post-analyse" et fournissent cette passe comme analyse auxiliaire intégrée au reste du framework. Il est donc difficile d'utiliser un slicer en dehors du framework dans lequel il a été développé.

Parmi les outils existants, [GUEB](#) [120] a été exclu, car il n'est plus utilisable ([Binsec](#) ayant beaucoup évolué depuis) et, car il est exclusivement centré sur les use-after-free. Un outil fortement considéré fut [Frama-C](#) [209] avec son plugin de slicing permettant de générer des [PDG](#). Cependant, celui-ci n'est plus maintenu et le coût d'une intégration de [Frama-C](#) dans la combinaison d'analyse serait vainement complexe.

[CodeSurfer](#) [185] est sans nul doute le plus ancien et le plus avancé des slicers (*bien qu'il ne soit pas libre*) il a donc été retenu pour l'étude. De même, [Symbiotic](#) [323] et plus particulièrement son slicer `dg` a été retenu, car pouvant s'intégrer sur toute combinaison basée sur [LLVM](#). L'outil [Chopper](#) [345], bien que combinant exécution symbolique avec [KLEE](#) et slicing, a été exclu, car il utilise `dg` comme dépendance externe. [Chopper](#) n'étant pas à l'origine de `dg`, le choix s'est donc prioritairement tourné vers [Symbiotic](#). L'approche n'en reste pas moins tout à fait intéressante. Le tableau 16.1 détaille les différentes caractéristiques des deux outils retenus.

La base de tests n'étant pas spécifiquement conçue pour tester les performances d'un slicer et comme un seul des deux retenus a pu être testé, les résultats sont directement donnés dans la section de description de l'outil (cf. 16.2). Ainsi, aucun chapitre dédié à l'évaluation des outils n'a été fait dans cette partie.

			#1 CodeSurfer	#2 Symbiotic
CTO	Général outils	CTO_1 : Langage	C, C++	C++
		Source	✓	✓
		x86	n/a	n/a
		x86-64	n/a	n/a
		ARMv7	n/a	n/a
		ARMv8	n/a	n/a
		CTO_3 : Open-source	✗	✓
CTO_S	Caractéristiques	CTO_4 : Licence	Payant	Libre
		CTO_5 : Documentation	/	✗
		CTO_6 : Activité	n/c	✓
		CTO_7 : Jeu de tests	/	/
		CTO_S1 : Type	statique dynamique	✓ ✗
		CTO_S2 : Direction	avant arrière	✓ ✓
		CTO_S3 : Exécutabilité	compilable logique	n/c n/c
		CTO_S4 : Dicing		✗
		CTO_F5 : Chopping		✗

TABLE 16.1 – Comparaison des outils de slicing sélectionnés.

## 16.1 Slicing #1 : CodeSurfer

CodeSurfer est une outil développé par la société GrammaTech<sup>168</sup> depuis sa création en 1988 par Tim Teitelbaum et Thomas Reps au sein de l'université de Cornell. Thomas Reps ayant largement publié sur les techniques de slicing, cet outil est une référence. Son successeur CodeSonar fonctionne de manière indifférenciée sur le source ou le binaire.

Cet outil est l'un des plus connus dans le monde industriel, car c'est un pionnier dans le domaine de l'analyse de programme, et pour cause, il capitalise sur plus de 30 ans de développement et d'évolution. L'équipe TECHx de GrammaTech ayant concouru au Cyber Grand Challenge s'est distinguée en terminant à la 2<sup>ème</sup> position<sup>169</sup>. Cependant rien n'atteste de l'utilisation de slicing pour la compétition.

Peu de documentation est disponible concernant le fonctionnement de l'outil, celui-ci étant payant. Cependant, les publications académiques à son sujet donnent un bon aperçu du fonctionnement interne (probable) de l'outil. Cette étude s'appuie sur plusieurs publications [17, 185] et sur une documentation de l'API qu'il a été possible de consulter.

### 16.1.1 Fonctionnalités

CodeSurfer fournit un grand nombre de fonctionnalités aussi bien sur du code source que sur du binaire. En termes de slicing, l'algorithme fonctionne statiquement et s'appuie sur les graphes de dépendances. Les slices peuvent être effectués aussi bien en avant, qu'en arrière, et il est aussi possible d'effectuer des "coupes" (*chop*) en fournissant une source et une destination. L'algorithme s'appuie sur le SDG dont l'algorithme est donné ci-dessous (16.1.2). L'outil fournit également un mécanisme de *scripting* et des fonctionnalités de "requêtage" et de visualisation.

### 16.1.2 Algorithme de calcul du SDG

L'algorithme de calcul du SDG à partir de code C est le suivant :

- application d'un préprocesseur C (custom) maintenant un lien étroit avec les sources originales ;
- parsing du résultat pour obtenir l'AST ;
- création d'un CFG dont chaque noeud est annoté avec les ensembles de *def-use* et de *possible-kill* calculés sans analyse de pointeur ;
- création d'une première approximation du graphe d'appel CG (sans analyse des appels de fonctions indirects/calculés) ;
- analyse des variables globales (sans analyse de pointeurs) ;

---

168. <https://www.grammatech.com>

169. <https://www.grammatech.com/cyber-grand-challenge>

- analyse des appels indirects de fonction et étend le graphe avec les cibles de saut potentielles ;
- analyse de pointeur qui enrichit les ensembles *def-use*, *possible-kill* et les variables globales qui sont associés à chaque nœuds du CFG ;
- construction du **PDG** par dépendance de contrôle et de données ;
- optionnellement, compression des régions fortement connectées d'un **PDG** par un seul nœud ;
- agrégation des **PDGs** pour former le **SDG** ;
- calcule des *summary-edges* pour chaque procédure et les lient entre elles ;
- écriture des différents graphes dans des fichiers ainsi que le lien qu'ils ont avec le code source.

### 16.1.3 Slicing au niveau binaire

**CodeSurfer/CodeSonar** est maintenant en mesure d'effectuer du slicing sur du binaire. La partie parsing du C et construction de l'**AST** est remplacée par un désassemblage du programme et une résolution des sauts indirects et calculée. Le schéma 16.1 montre le prototype réalisé en 2005 [17] se basant sur **IDA Pro**.

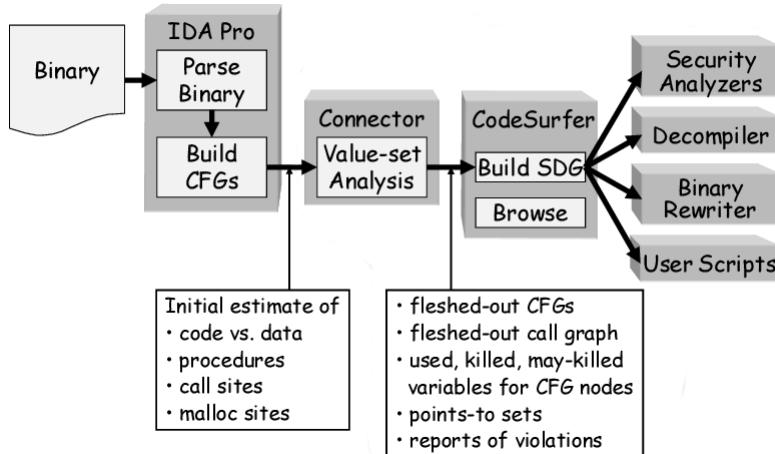


FIGURE 16.1 – Architecture des composants de slicing au niveau binaire (src. [17]).

Il est intéressant de noter que l'algorithme de calcul du **SDG** se base entièrement sur une analyse **VSA** par interprétation abstraite (cette analyse ayant elle aussi été inventée par Thomas Reps [18]). Elle permet d'obtenir l'ensemble des valeurs pour chaque variable et notamment de déduire les *points-to sets* sur les pointeurs (*pour les sauts calculés*). À partir de ces informations, le **SDG** peut être calculé sur le CFG avec le même algorithme que pour le source.

### 16.1.4 Utilisation

Bien qu'il n'ait pas été possible de le tester, **CodeSurfer** fournit trois modes de slicing à partir d'un critère de slice appelé *query-point* dans ce contexte :

- **Across and In** suit les fonctions appelées ;
- **Across and Out** suit les fonctions appelantes ;
- **Full** suit les fonctions appelées et les fonctions appelantes.

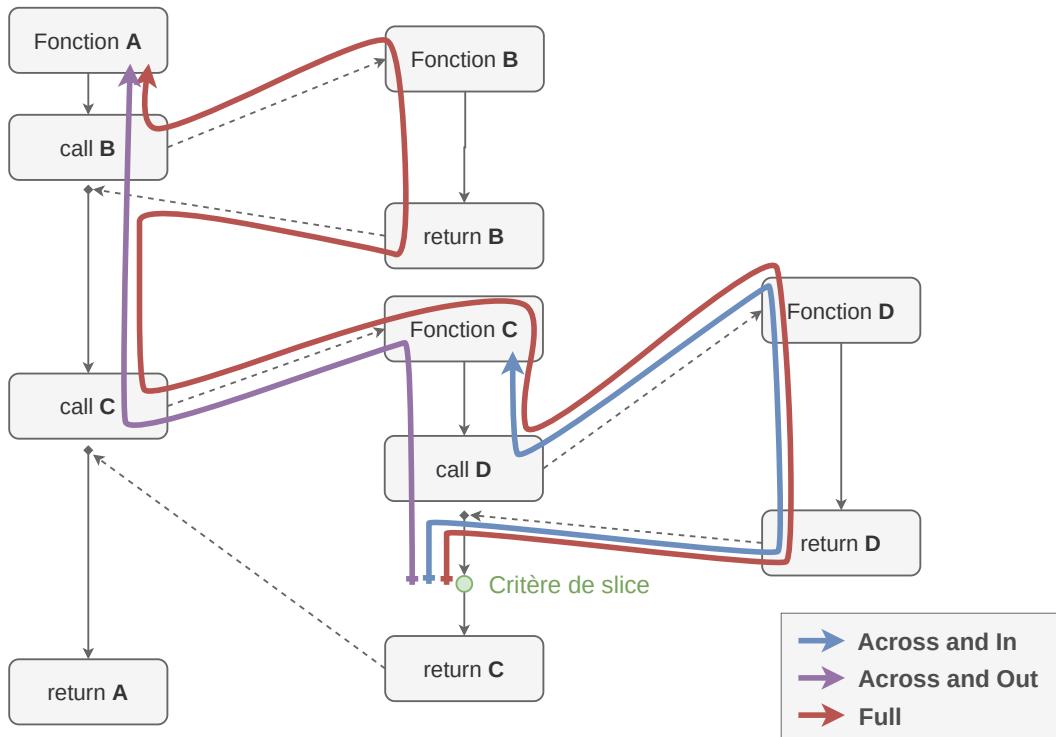


FIGURE 16.2 – Mode de slice de **CodeSurfer**.

Le schéma 16.2 donne un exemple du slice en arrière emprunté en fonction des trois modes. Le critère de slice se trouve dans la fonction C, qui appelle la fonction D et qui est elle-même appelée par A. Dans cette configuration, chacun des trois modes fournit un slice différent. Par défaut les slices sont interprocéduraux (mais cela peut être désactivé avec `-compute-summaries=no`). Il est aussi possible de ne suivre que les dépendances de contrôle où que les dépendances de données.

## 16.2 Slicing #2 : Symbiotic

Symbiotic est un outil combinant instrumentation, slicing et exécution symbolique pour la vérification de programme. C'est l'un des seuls outils à fournir un slicer open-source et potentiellement utilisable en-dehors de l'outil. C'est pour cette raison que cette ressource a été choisie pour être étudiée. Le slicer nommé dg<sup>170</sup> fonctionne sur le bit-code LLVM tout comme l'instrumentation dynamique appelée sbt-instrumentation<sup>171</sup> et l'exécuteur symbolique, KLEE [56]. Tous ces composants sont assemblés dans le framework Symbiotic<sup>172 173</sup> et publié sur Github.

L'ensemble du framework semble avoir une activité de développement régulière. La figure montre le graphe d'activité de dg maintenu en continu depuis février 2015. Symbiotic et dg comptent respectivement 1285 et près de 2000 commits.



FIGURE 16.3 – Graphe d'activité des commits de dg.

Bien que ce framework ait une visée de vérification logicielle et pas de recherche de vulnérabilité, la combinaison des techniques est proche de ce qui pourrait être fait dans PASTIS pour COMB\_SICF. L'étude s'appuie majoritairement sur l'article suivant [323] :

“Checking Properties Described by State Machines : On Synergy of Instrumentation, Slicing, and Symbolic Execution” par Jiri Slaby, Jan Strejcek et Marek Trtík publié au 17th International Formal Methods for Industrial Critical Systems Workshop, FMICS, 2012

### 16.2.1 Approche

Les travaux visent à vérifier des propriétés sur des machines à état finies (FSM) en se basant sur le fait que les outils existants produisent trop de faux positif. L'approche se veut donc sans faux positifs pour l'exécution symbolique et s'effectue en trois étapes gérées par trois composants :

170. <https://github.com/mchalupa/dg>

171. <https://github.com/staticafi/sbt-instrumentation>

172. <https://github.com/staticafi/symbiotic>

173. <http://staticafi.github.io/symbiotic/>

**Prepare** : instrumentation du programme, avec ajout de code pour suivre le comportement de la **FSM** (et localiser des verrous `lock`, `unlock` dont l'article veut s'assurer qu'ils ne mettent pas la machine dans un état impropre).

**Slicer** : slicing interprocedural du programme avec comme critère de slice les variables en mémoire représentant les états de la **FSM**. De cette manière, seul le code de la machine à état est conservé dans le slice. Le slicing s'appuie sur l'analyse *points-to* pour les dépendances de données.

**Kleerer** : exécution symbolique du slice pour vérifier l'atteignabilité de ces états d'erreur.

Il eut été intéressant de détailler le fonctionnement de ces trois composants, mais l'article ne donne aucune information supplémentaire. Les expérimentations ont apparemment été faites sur des fonctions du noyau Linux sans plus de détails sur comment elles ont été obtenues. KLEE ne supportant pas l'exécution symbolique de code noyau, celles-ci ont forcément été extraites dans un programme *userland*. Au-delà d'un papier d'une qualité passable et d'expériences très très discutables quant à leur pertinence, le code du slicer lui, semble digne d'intérêt.

### 16.2.2 Slicer dg

Le fonctionnement des différents algorithmes de **dg** (pour *dependence graph*) a très bien été documenté dans un rapport de thèse dédié au slicing de bitcode LLVM [66]. **dg** n'a aucune autre dépendance que **LLVM**. Toutes les analyses de *def-use* et de *points-to* sont donc implémentées en interne et à la compilation du projet. Il se présente sous la forme d'une bibliothèque et d'un certain nombre d'utilitaires utilisant cette bibliothèque :

- **llvm-slicer**, binaire principal effectuant le slicing ;
- **llvmdg-show**, export du graphe de dépendance en graphviz (avec **llvmdg-show** pour une conversion directe en pdf) ;
- **llvm-ps-show**, export du graphe de pointeur résultant de la Points-To Analysis (**PTA**) (avec **ps-show** pour une conversion en pdf) ;
- **llvm-rd-dump**, export du graphe de *reaching-definition* (avec **rd-show** pour une conversion en pdf) ;
- **llvm-vr-dump**, export du graphe de *value-relation* ;
- **llvm-to-source**, affiche les lignes de code source toujours dans le slice si le programme a été compilé avec **-g**.

Le plus important est **llvm-slicer** car il effectue le slicing à proprement parler. Les autres génèrent des graphes à titre informatif ou de debug. **dg** est spécialement conçu pour effectuer du slicing en arrière.

#### Utilisation et test de **llvm-slicer**

L'argument le plus important de l'outil est le critère de slicing (option **-c**) qui doit être un nom de fonction ou bien une ligne et une variable dans le code source si celui-

ci a été compilé avec `-g`. Plusieurs autres options permettent de configurer en détail le comportement de l'algorithme :

- `-cd-alg=[classic, ce]`, définit l'algorithme de dépendance de contrôle à appliquer entre l'algorithme de Ferrante [121] et un algorithme expérimental sur les expressions de contrôle ;
- `-entry`, nom de la fonction à considérer comme point d'entrée ;
- `-forward`, effectue le slicing en avant ;
- `-preserved-functions`, protège des fonctions à ne pas slicer ;
- `-pta=[fi, fs, inv]`, sensibilité de l'analyse de pointeurs (cf. 15.3.2), avec *fi* (flow-insensitive) et inversement *fs* (flow-sensitive) ;
- `-rda=[dense, ss]`, algorithme de Reaching Algorithm Analysis (RDA).

Le listing 27 montre le slicing de l'exemple 26 avec la fonction `add` comme critère de slice. Le graphe 16.4 montre le graphe de dépendance généré pour le calcul du slice. Cela donne un aperçu de la complexité de ce genre de graphe même pour des exemples très simples.

---

**Listing 27** Slicing de l'exemple de 26 avec dg

---

```
1 clang -c -emit-llvm exemple.c -o exemple.bc
2 #llvm-link ... (si plusieurs .bc doivent être liés)
3
4 llvm-slicer -c add exemple.bc -o exemple.sliced
5
6 # compilation du slice
7 llc -filetype=obj exemple.sliced -o exemple.o
8 gcc -o exemple exemple.o
```

---

En termes de gain, un simple parcours de l'AST LLVM du programme avant et après slicing permet d'avoir un aperçu de la taille du slice. L'extrait de code donné en Listing 28 permet de parcourir le fichier .bc (donné comme argument du programme) et de calculer le nombre de fonctions, basic-blocks et instructions du programme. Pour l'exemple de test, le code passe de 3 fonctions, 5 basic-block et 36 instructions à 2 fonctions pour 25 instructions. Le slice est donc environ 30% plus compact. Pour de plus gros exemples, il est vraisemblable que le gain soit beaucoup plus significatif. Ce résultat est purement indicatif et nécessiterait un benchmark spécifique pour évaluer le gain en temps et en couverture du programme.

## 16.2. Slicing #2 : Symbiotic

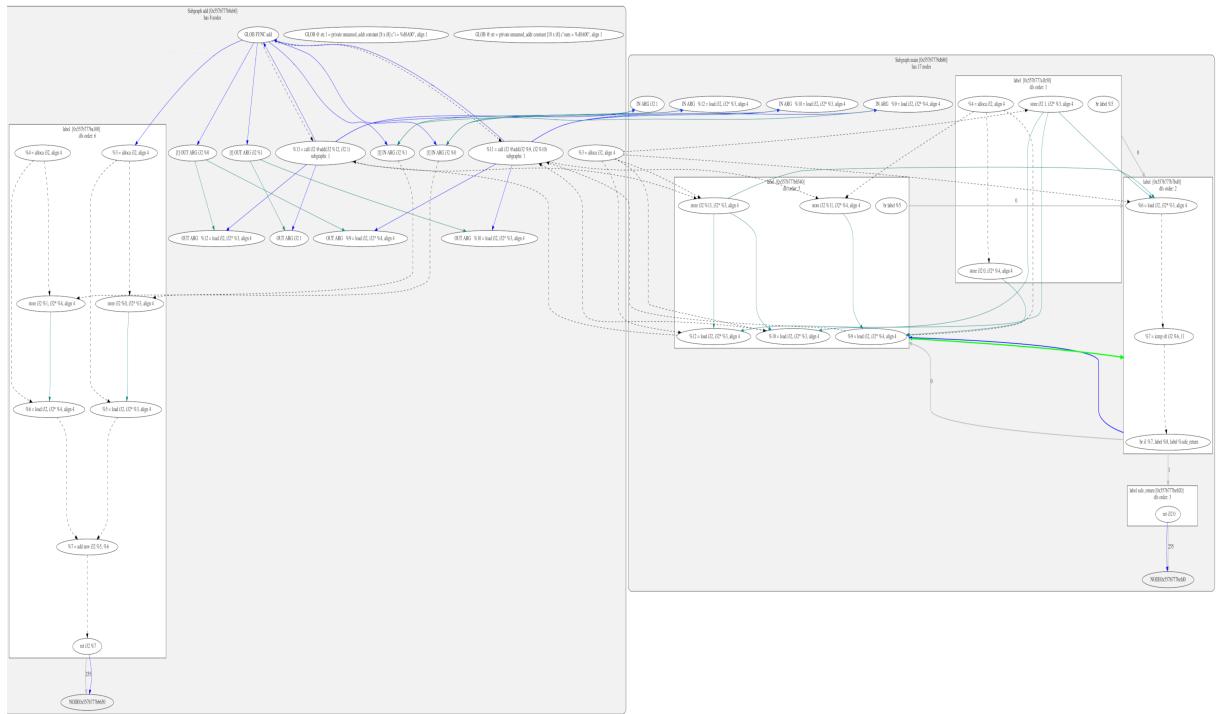


FIGURE 16.4 – Graphe de dépendance du slice de 26.

**Listing 28** Exemple de parcours de l'AST LLVM en Python

```

1  #!/usr/bin/env python3
2  import sys
3  from llvmpy.llvm import *
4
5  buffer = create_memory_buffer_with_contents_of_file(sys.argv[1])
6  context = get_global_context()
7  module = context.parse_ir(buffer)
8
9  n_funcs, n_bb, n_inst = 0, 0, 0
10
11 for function in module.iter_functions():
12     n_funcs += 1
13     for bb in function.iter_basic_blocks():
14         n_bb += 1
15         for instruction in bb.iter_instructions():
16             n_inst += 1
17
18 print("Funs:{}, BB:{}, Inst:{}".format(n_funcs, n_bb, n_inst))

```

## 16.3 Synthèse & Conclusion

L'outil **dg** fonctionne très bien. Il est simple et limpide d'utilisation. Cependant, aucun test n'a été effectué pour évaluer la pertinence des slices générés. Cela n'a pas été fait faute de temps et parce qu'il aurait fallu benchmarker précisément le temps pris par chaque outil sur chaque test.

En termes de combinaison, il n'y a aucune intelligence dans la combinaison de **dg** et de **KLEE** dans **Symbiotic** puisque **KLEE** se contente de prendre en entrée le programme slicé. En revanche, **Chopper** [345] semble interconnecter les deux de manière astucieuse pour tirer le meilleur profit du slice dans le DSE. Bien qu'il n'ait pas été possible de le tester (faute de temps) cette approche est une source d'inspiration pour le projet PASTIS.

Pour une combinaison **COMB\_SICF**, l'utilisation de **dg** sous forme de bibliothèque est très certainement faisable. Néanmoins, cela a le principal revers d'obliger les outils de fuzzing DSE à se baser sur **LLVM**. Une alternative serait d'extraire le graphe de dépendance et d'être en mesure de l'utiliser en faisant le lien **LLVM** → assembleur (dans le binaire). En effet, générer autant de programmes (slices) que d'alertes levées par **Klocwork** serait une mauvaise idée, car il empêcherait tout partage de connaissances entre les instances du programme qui partagent forcément des portions de code et des préfixes de chemins identiques. Les fuzzers et DSE repartiraient à "zéro" pour chaque slice.

La solution la plus intéressante en termes de slicing serait de calculer des slices pour chaque objectif de test, mais de ne les appliquer que de manière logique au fuzzer et au DSE pendant la couverture (à la manière de **Chopper** [345]). Ceux-ci se concentreraient donc sur un seul et même binaire capitalisant sur les chemins et entrées découverts. L'inconvénient est que cela implique de modifier le code de sélection/élagage de branches pour le DSE et pour le fuzzing. Cela implique aussi d'en avoir une bonne connaissance. Cette méthode semble néanmoins la plus viable, tant en termes d'efficacité que de performance.

# **Septième partie**

## **Conclusion**



## Chapitre 17

---

# Propositions de combinaisons

---

En se basant sur l'étude des ressources, l'observation et l'analyse de leurs résultats, plusieurs méthodes de combinaisons se démarquent. Dans le cadre du projet, les considérations suivantes ont été prises en compte pour les propositions :

- le code source est disponible, la combinaison doit en tirer profit au mieux ;
- la priorité est donnée aux outils dont le développement est actif, ayant une certaine assurance du support futur et (de préférence) dont Quarkslab a une connaissance suffisante ;
- la priorité est donnée aux outils faciles à prendre en main ;
- la priorité est donnée aux outils pouvant être combinés facilement (donc fournis sous forme de bibliothèques ou étant écrits dans un langage usité).

Conformément aux exigences du projet, les sections ci-dessous décrivent les propositions de combinaisons mêlant analyse statique, fuzzing, exécution symbolique et slicing. Bien que cela n'ait pas été vérifié, toutes les combinaisons proposées ci-dessous sont théoriquement compatibles avec la maquette (vraisemblablement une carte **NXP i.MX 8**<sup>174</sup>). En effet, la carte est en mesure de faire tourner un système Linux, *a priori* compatible avec les différents composants impliqués dans les différentes combinaisons. Évidemment, comme ce fut le cas pour **Honggfuzz** avec les tests sur ARM, il est très probable que de petites incompatibilités ou bugs soient rencontrés durant la phase de configuration et de tests.

Les trois propositions sont des ébauches de ce qu'il serait possible de faire au vu des tests effectués. Celles-ci sont loin d'être définitives et de nombreux détails de conception restent à définir. La solution retenue sera raffinée et développée dans le premier livrable du poste 2.

---

174. <https://www.nxp.com/docs/en/fact-sheet/IMX8FAMFS.pdf>

## 17.1 Interfaçage avec Klocwork

L’analyse statique pour les trois combinaisons considérées s’appuie sur celle de **Klocwork** dont le but est de fournir les objectifs de test et par la même occasion les critères de slice. Les alertes levées seront converties en expressions en langage C (*d’une manière qu’il reste à définir*) puis ajoutées sous forme d’assertions (spécifique à l’outil choisi e.g., `klee_assert` ou `triton_assert`). Ces assertions seront compilées dans le binaire final et donc facilement identifiables dans celui-ci (par l’appel de fonction). Pour une même alerte **Klocwork**, une ou plusieurs assertions pourront être encodées pour tester la présence du bug et pour tester son exploitabilité.

Par ailleurs, il n’est pas exclu d’utiliser d’autres analyses statiques auxiliaires, comme de l’analyse de valeurs par interprétation abstraite avec **IKOS** [46], ce qui permettrait de faire du DSE précontraint (comme **Mayhem** [64]). Néanmoins, cela serait fait de manière tout à fait opportuniste en complément de **Klocwork**.

## 17.2 Proposition #1

La première combinaison s’articule autour de **Klocwork**, **Honggfuzz** [335] et **Triton** [304]. Au vu des résultats, **Honggfuzz** est indéniablement le fuzzer le plus adapté au projet PASTIS. Il est activement développé et le support du fuzzing de protocoles réseau, bien qu’expérimental, est activement développé comme en atteste la bibliothèque interne appelée `libhfnetdriver`<sup>175</sup>. Enfin, c’est le seul à supporter nativement le fuzzing sur ARM ce qui en fait le fuzzer idéal.

Dans cette combinaison, à la manière du DSE dit sélectif [333], **Triton** recevrait du fuzzer des chemins jugés difficiles à résoudre. Sur celui-ci, il ferait la propagation de teinte et l’exécution symbolique afin de générer une nouvelle entrée qui serait ajoutée à la liste des états (configurations actives) du fuzzer.

Ce modèle pourrait s’implémenter selon les deux modes suivantes :

- *embarqué* : la bibliothèque **Triton** serait directement “linké” à **Honggfuzz** pour effectuer les interactions directement en mémoire ; les deux composants tourneraient dans le même processus.
- *séparé* : **Triton** est compilé dans un processus à part qui interagit en client serveur avec le fuzzer ; celui-ci envoie une requête à **Triton** pour résoudre un prédicat de chemin qui l’exécute symboliquement (statiquement) puis renvoie l’entrée générée au fuzzer (s’il est satisfiable).

La première méthode évite les communications entre les deux composants, mais oblige à tout exécuter sur la carte de test. Le deuxième mode permet de déporter l’exécution symbolique sur l’ordinateur connecté à la carte via le lien IP (persistant) du noyau Linux. De

---

175. <https://github.com/google/honggfuzz/tree/master/libhfnetdriver>

surcroît, les pertes de performances des communications sont définitivement négligeables en comparaison du temps de résolution des formules SMT et du gain de performance lié à l'exécution sur un processeur de bureau (Intel Core i7 etc.).

À ce modèle pourrait éventuellement être ajouté `dg`, qui en phase de pré analyse, générera un slice pour chaque objectif de test. Un graphe de dépendance serait alors généré pour chaque objectif de test. La fonction de “fitness” (cf. 9.7.2) de `Honggfuzz` devrait alors être modifiée pour favoriser les configurations restant dans le slice. Pour éviter de lancer une instance du fuzzer par objectif de test, il faudrait modifier le fonctionnement du fuzzer pour les ordonner et essayer de les couvrir en partageant le temps passé sur chacun d'entre eux. À noter qu'adapter la fonction de fitness est loin d'être trivial et la stratégie visant à s'approcher de l'objectif de test rentrerait en conflit avec l'algorithme de mutation génétique qui récompense les nouvelles portions de code couvertes.

## 17.3 Proposition #2

Cette proposition vise à combiner `Honggfuzz`, `KLEE` et optionnellement `dg`, pour le slicing. Le choix de `Honggfuzz` est motivé par les mêmes raisons que la proposition #1. Le choix de `KLEE` est motivé par sa rapidité et sa robustesse. À l'exception des performances sur LAVA-M, les résultats étaient très satisfaisants. L'insertion des objectifs de test de `Klocwork` se ferait via la fonction intrinsèque `klee_assert` de `KLEE` et un traitement particulier devrait leur être donné au niveau du fuzzer.

L'idée de cette combinaison est de faire fonctionner les deux outils de manière asynchrone en les faisant communiquer indirectement par leurs dossiers de travail (*workspace*) respectif. Chacun essayerait donc de couvrir les objectifs de test indépendamment. Cette combinaison nécessiterait à minima l'implémentation d'une glue d'interface entre les formats utilisés par l'un et l'autre des outils.

Utilisant la même représentation intermédiaire, le LLVM IR, il serait relativement aisé de combiner `KLEE` avec `dg`. Dans ce contexte, l'idée serait de les combiner “à la Chopper [345]” pour que le DSE tire réellement profit du slicing. Pour rappel, `Chopper` n'exécute symboliquement que ce qui est inclus dans le slice, le reste est évalué de manière “fainéante” sans spécifiquement essayer de couvrir ou de trouver des bugs. `Chopper` ayant fait des modifications dans `KLEE` pour exposer des fonctions d'API de slicing au DSE et pour ajouter le slicing de fonctions arbitraires dans `dg`, il serait intéressant d'étudier la possibilité d'utiliser directement `Chopper` (*bien qu'il relève plus du PoC pour le moment*).

Le principal inconvénient d'une telle combinaison serait de se familiariser et de rentrer dans le code de `KLEE`, `Chopper` et `dg` qui implémentent de nombreuses fonctionnalités et optimisations accumulées au fil du temps. Un autre inconvénient est que, hormis les entrées qui seraient partagées entre les deux outils, aucune autre information concernant la couverture ou autre ne pourrait être partagée facilement.

Inversement, un avantage significatif avec le DSE de `KLEE` est qu'il est indépendant de l'architecture cible. De cette manière, il serait aussi possible de faire fonctionner `KLEE`

sur une machine déportée et Honggfuzz sur la carte, les deux ayant alors leurs ressources propres.

## 17.4 Proposition #3

La troisième proposition est fortement inspirée du fonctionnement de `Mayhem` [64] et de `Qsym` [380] qui partagent des similarités. Cette combinaison s'appuie sur `Honggfuzz` pour le fuzzing (en lieu et place du AFL de `Qsym`) et sur `Qsym` pour la partie exécution symbolique. Le problème est que l'instrumentation de `Qsym` se base sur `Pin` [241] (purement Intel-x86) et qui plus est, l'exécution symbolique est entièrement basée sur x86 (sans aucune chance de portage puisque qu'aucune IR n'est utilisée). L'idée est donc de remplacer l'instrumentation dynamique (`DBT`) par `QBDI` [181], la `DBT` développée à Quarkslab<sup>176</sup>, puis de remplacer l'exécution symbolique et la propagation de teinte (faite par `libdft` [204]) par `Triton` qui effectue les deux. Celui-ci fait par conception de l'exécution symbolique granulaire préconisée par les auteurs de `Qsym`. L'idée est, autant que possible, de capitaliser sur l'approche de DSE relaxée de `Qsym` et de toutes les optimisations qui ont été faites comme le *optimistic solving* ou le *basic block pruning* (par regroupement d'exécution ou par sensibilité au contexte) etc.

L'architecture de `Qsym` ressemble beaucoup à celle de `Mayhem` et tous deux ont fait leurs preuves en termes d'efficacité. `Qsym` et `Mayhem` possèdent une instrumentation dynamique (`DBI`) effectuant l'exécution concrète et la propagation de la teinte. Ce composant s'appelle *Concrete Executor Client* (CEC) dans `Mayhem`. C'est lui qui est chargé de transmettre l'ensemble d'instructions teintées à exécuter symboliquement (par le *Symbolic Executor Server* SES). Dans `Qsym`, cela est effectué au sein même de l'instrumentation (dans le pintool), mais c'est en essence le même fonctionnement. Cette combinaison s'inspire donc des deux approches.

Les composants `Honggfuzz` et “`Qsym-like`” interagiraient ensuite, comme cela a été fait dans `Qsym` avec AFL. Tout comme la proposition #2, l'interaction s'effectuerait via les répertoires de travail. Ces deux composants faisant tous deux de l'instrumentation dynamique, il sera nécessaire de les faire fonctionner sur la carte ARMv8. Deux cas de figure se présentent pour l'implémentation du composant `Qsym-like` :

- *cas favorable* : le code de `Qsym` est suffisamment modulaire et permet de remplacer `Pin` par `QBDI`, tout en insérant `Triton` pour le “lifting” des instructions en SMT ; les composants d'analyse pourront à quelques modifications près fonctionner de la même manière de façon transparente.
- *cas défavorable* : le code de `Qsym` n'est pas portable ; dans ce cas il faut réécrire la logique de `Qsym` sur `QBDI + Triton` (*ce qui peut être, en définitive, plus simple*).

Les auteurs de `Qsym` promeuvent la suppression d'une IR au profit des performances. Au-delà du fait que le gain ne vaut certainement pas la perte de la portabilité, insérer

---

176. `QBDI` ne supporte pas encore ARMv8 (ce qui est dans le planning des futurs développements).

Triton ne fait pas perdre ce gain. En effet, `Qsym` transforme directement les instructions teintées en formules SMT. Or, c'est exactement ce que fait `Triton` puisqu'il transforme les instructions dans une représentation interne très proche du SMT.

De la même manière que la proposition #1, un slicing léger pourrait être introduit dans la combinaison. Il se baserait aussi sur le `SDG` représentant les dépendances de données et de contrôle du programme généré par `dg`. Ce slicing pourrait donner lieu à des optimisations intéressantes, par exemple en appliquant uniquement le *optimistic-solving* aux contraintes issues du slice.

Les performances et l'implémentation de cette combinaison sont très incertaines car de nombreuses incertitudes restent à lever. Néanmoins, si celle-ci fonctionne comme attendu en fournissant les mêmes performances que `Qsym` sur x86, il est certain que ce serait le meilleur fuzzer whitebox existant pour ARMv8.

## 17.5 Fonctionnalités complémentaires

La combinaison sélectionnée jouera un rôle essentiel dans les performances obtenues par le démonstrateur. Cependant, de nombreuses optimisations et astuces sont développées dans des articles de recherche ou développées dans des outils à l'état de l'art. Sans aller jusqu'aux extrêmes tels que la linéarisation de contraintes de `Mayhem` [64], plusieurs optimisations pourraient être implémentées dans la mesure du possible dans le démonstrateur pour améliorer ses performances.

Loin de s'engager à les intégrer, voici plusieurs optimisations qu'il serait intéressant de considérer dans le démonstrateur si le planning de développement le permet.

**Stratégie de recherche dirigée.** Le but de l'exécution du démonstrateur n'est non pas de couvrir tout le programme à la recherche de vulnérabilités, mais uniquement couvrir les objectifs de test générés par `Klocwork`. Ainsi, il serait judicieux d'implémenter des stratégies de recherche dirigée comme SDSE [243] pour le DSE (cf. 6.7.2). Appliquées à du fuzzing, cela consistera à adapter la fonction de "fitness" pour favoriser les entrées rapprochant la couverture de la cible à couvrir.

**Optimisation de l'instrumentation.** Les travaux très récents autour de `UnTracer-AFL` [262] et des optimisations qu'ils ont effectués sur l'instrumentation dynamique sont très prometteurs. Ils s'appuient sur deux observations : 1) seule une fraction des entrées générées exercent un nouveau chemin, 2) la rapidité de l'augmentation de la couverture décroît au fur et à mesure. Ainsi, l'idée et de ne tracer dynamique que les entrées dont il est sûr qu'elles améliorent la couverture. Cela évite de tracer toutes les entrées qu'elles améliorent ou non la couverture. Des gains significatifs ont été montrés avec `Qsym`. Il serait donc intéressant d'employer cette technique si la proposition #3 est retenue.

**Caching & Snapshots.** Les outils avancés comme `Mayhem` ou `KLEE` utilisent différentes structures de données pour mettre en cache, les états, les formules, les résultats de formule etc. L’impact est loin d’être négligeable et il peut faire la différence. Par exemple, `GRR` (la `DBI` de `cyberdyne`) utilise un cache (persistent) du code instrumenté qui permet de relancer autant d’exécutions sans jamais ré instrumenter le programme (et sans “forker”). Cela implique d’avoir une connaissance parfaite de l’agencement mémoire du programme.

**Élagage des vérifications non critique.** Plusieurs travaux [54, 276] montrent que les sommes de contrôle (*checksum*) ralentissent considérablement et inutilement les fuzzers et outils de DSE. Ces travaux proposent différentes méthodes pour les détecter et les outrerpasser. Les travaux récents de Mathias Payer en 2018 dans `T-Fuzz` [276] étendent ces vérifications avec le terme NCC (*Non Critical Check*). L’auteur propose une méthode pour les détecter, les désactiver pour le fuzzing puis les recalculer sur l’entrée finale pour générer des fichiers valides. Le programme étalon implémentant une pile IP complète ces *checksums* seront très nombreux et vont invariablement nuire aux performances. Il serait donc très judicieux d’implémenter cette optimisation dans le démonstrateur. `Qsym` reconnaît ce travers dans son approche et suggère l’approche de `T-Fuzz` comme axe d’amélioration.

Ces quelques exemples, loin d’être exhaustifs, donnent un aperçu de quelques optimisations qu’il serait opportun d’implémenter dans le démonstrateur si les outils et le temps alloué au développement s’y prêtent.

## Chapitre 18

---

### Conclusion Générale

---

Cet état de l'art s'est efforcé de couvrir le plus large panel de publications et d'outils allant du premier article sur l'exécution symbolique par James C. King en juillet 1976 [208] à l'approche combinée de fuzzing et d'exécution symbolique d'**Eclipser** qui sera publiée le 31 mai 2019. Évidemment, ne pouvant être exhaustifs, de nombreux articles et outils ont été laissés de côté. Alors que la recherche en slicing souffre d'une certaine inertie, le fuzzing, jouit d'un incroyable dynamisme. Entre le début et la fin de cette étude, pas moins d'une trentaine d'articles scientifiques ont été publiés présentant de nouvelles approches et améliorations aux techniques de fuzzing actuel.

Malgré les contestations possibles concernant le choix de la suite de test, les résultats n'en restent pas moins très instructifs quant aux capacités des outils. Les résultats sont à l'évidence, les résultats les plus complets de la littérature avec 10 outils testés, et dont certains n'avaient jamais été comparés entre eux, en particulier **Angora**, **Qsym** et **Eclipser**. Ces résultats entérinent l'utilité de combiner les techniques et dénotent une nette amélioration en termes de résultats sur les outils les plus récents de l'état de l'art (e.g., **Qsym**, **Angora**, **Eclipser**).

Outre la nécessité de combiner les approches, la tendance qui se dégage est le besoin de se libérer du DSE “orthodoxe” où la correction prime par dessus tout. Les approches qui passent le mieux à l'échelle favorisent maintenant une analyse de teinte performante, du DSE relaxé (potentiellement incorrect) ou encore du “DSE” sans formule SMT basée sur des contraintes sur un domaine d'intervalle. En sus, sous une autre forme, le slicing connaît un renouveau avec les techniques récentes de recherche de vulnérabilités à base de teinte statique calculée sur des “Property Graph”[373] et publié récemment en 2018 [148]. Tous ces axes de recherche et toutes ces tendances ont fourni de nombreuses idées en vue de l'élaboration de la combinaison d'analyses.

Les deux finalistes de la compétition du CGC, ForAllSecure et GrammaTech, capitalisent sur respectivement 20 et 30 ans de recherche. Leurs succès amplement mérités montrent l'engagement et la persévérance nécessaire pour créer des outils de ce niveau de sophistication. Loin de prétendre les concurrencer, il reste néanmoins une bonne marge de manœuvre pour créer une combinaison d'analyses qui dépasse l'état de l'art des outils

## *Chapitre 18. Conclusion Générale*

---

publics dans la recherche automatisée de vulnérabilités sur ARMv8 largement ignorée pour le moment.

---

# Acronymes

---

- ABI** Application Binary Interface. i, 135, 136, *Glossary* : Application Binary Interface
- ACFG** Augmented Control Flow Graph. i, 238, *Glossary* : Augmented Control Flow Graph
- AEG** Automated Exploit Generation. i, 28, 63, 101
- AI** Abstract Interpretation. i, 10, 232, 233
- AST** Abstract Syntax Tree. i, 48, 51, 109, 245, 246, 250, *Glossary* : Abstract Syntax Tree
- BB-DSE** Backward-Bounded DSE. i, 63, 64, 240, *Glossary* : Backward-Bounded DSE
- BFS** Breath-First-Search. i, 62, 103, *Glossary* : Breath-First-Search
- CAS** Center for Assured Software. i, 18
- CCTP** Cahier des Clauses Techniques Particulières. i, 153
- CFG** Control Flow Graph. i, 63, 140, 233, 238, *Glossary* : Control Flow Graph
- CFI** Control Flow Integrity. i, 140, *Glossary* : Control Flow Integrity
- CG** Call Graph. i, 63, 245, *Glossary* : Call Graph
- CGC** Cyber Grand Challenge. i, 11, 17, 19, 80, 85, 86, 124
- CoW** Copy-on-Write. i, 95, *Glossary* : Copy-on-Write
- CP** Constraint Programming. i, 66, 221
- CRS** Cyber Reasoning System. i, 19, 45, 80, 85, 100, 107, 191, 193, 197, 199, 213
- CWE** Common Weakness Enumeration. i, 23, *Glossary* : Common Weakness Enumeration
- DAG** Directed Acyclic Graph. i, 235, *Glossary* : Directed Acyclic Graph
- DARPA** Defense Advanced Research Projects Agency. i, 11, 19, 45, 199
- DBI** Dynamic Binary Instrumentation. i, 112, 258, 260, *Glossary* : Dynamic Binary Instrumentation
- DBT** Dynamic-Binary-Translation. i, 209, 214, 258, 268, *Glossary* : Dynamic-Binary-Translation
- DFG** Data Flow Graph. i, 140, *Glossary* : Data Flow Graph
- DFS** Depth-First-Search. i, 62, 103, *Glossary* : Depth-First-Search
- DGA** Direction Générale de l'Armement. i, 5
- DGA-MI** Direction Générale de l'Armement, Maîtrise de l'Information. i, 5
- DOS** Denial-of-Service. i, 121
- DSE** Dynamic Symbolic Execution. i, 45, *Glossary* : Dynamic Symbolic Execution
- DUA** Dead, Uncomplicated and Available. i, 23
- EA** Evolutionary Algorithm. i, 134, *Glossary* : Evolutionary Algorithm

## Acronymes

---

- EDB** Exploit DB. i, 22, *Glossary : Exploit DB*
- ETS** Error Transition System. i, 23
- EVM** Ethereum Virtual Machine. i, 86, *Glossary : Ethereum Virtual Machine*
- FCS** Fuzzing Configuration Scheduling. i, 133, 151, *Glossary : Fuzzing Configuration Scheduling*
- FSM** Finite State Machine. i, 63, 248, 249, *Glossary : Finite State Machine*
- GUI** Graphical User Interface. i, 101, 220
- IA** Intelligence Artificielle. i, 311
- IR** Intermediate Representation. i, 47–49, 51, 52, *Glossary : Intermediate Representation*
- ISA** Instruction Set Architecture. i, xv, 49, 309, 310
- JIT** Just-In-Time. i, 55, 209, *Glossary : Just-In-Time*
- LSTM** Long Short-Term Memory. i, 136, 144, *Glossary : Long Short-Term Memory*
- MDG** Method Dependence Graph. i
- MIT** Massachusetts Institute of Technology. i, 307
- MQTT** Message Queuing Telemetry Transport. i, 164, *Glossary : Message Queuing Telemetry Transport*
- NSA** National Security Agency. i, 18
- PASTIS** Programme d’Analyse Statique et de Tests Instrumentés pour la Sécurité. i, iii, 5–7, 11, 169, 174, 185, 186
- PDG** Program Dependency Graph. i, 232, 233, 235, 236, 243, 246, *Glossary : Program Dependency Graph*
- PHP** PHP Hypertext Preprocessor. i, 135, *Glossary : PHP Hypertext Preprocessor*
- PID** Process IDentifier. i, 119, 157, *Glossary : Process IDentifier*
- PoC** Proof-of-Concept. i, 22
- POV** Proof Of Vulnerability. i, 19, *Glossary : Proof Of Vulnerability*
- PTA** Points-To Analysis. i, 249, *Glossary : Points-To Analysis*
- RDA** Reaching Algorithm Analysis. i, 250
- RNN** Recurrent Neural Network. i, 136, *Glossary : Recurrent Neural Network*
- ROP** Return Oriented Programming. i, 28
- RTOS** Real Time Operating System. i, 239
- SARD** Software Assurance Reference Dataset Project. i, 18, 19, 224, *Glossary : Software Assurance Reference Dataset Project*
- SBE** Symbolic Backward Execution. i, 63, 64
- SDG** System Dependency Graph. i, 232–237, 239, 245, 246, 259, *Glossary : System Dependency Graph*
- SE** Symbolic Execution. i, 45
- SMT** Satisfiability Modulo Theories. i, 45, 311
- SSA** Static Single Assignment. i, 52, *Glossary : Static Single Assignment*
- VSA** Value Set Analysis. i, 58, 83, 246, *Glossary : Value Set Analysis*
- WCET** Worst-Case-Execution-Time. i, 239, *Glossary : Worst-Case-Execution-Time*
- WP** Weakest-Precondition Calculus. i, 10, 11, 61, 63, 65, *Glossary : Weakest-Precondition Calculus*

**XML** Extensible Markup Language. i,  
135, 175, *Glossary : Extensible Mar-*  
*kup Language*

*Acronymes*

---

---

# Glossaire

---

**Abstract Syntax Tree** Un arbre de syntaxe abstraite est un arbre dont les nœuds sont les opérateurs/instructions et les feuilles les opérandes. i, 48, 263

**adresse de retour** L'adresse de retour désigne l'instruction ajouté sur la pile avant l'appel à une fonction. Cette adresse est généralement l'instruction située immédiatement sous le `call`. i

**Application Binary Interface** Interface bas-niveau de communication entre le noyau et les programmes utilisateurs. i, 135, 263

**Augmented Control Flow Graph** CFG dans lequel est ajouté des arêtes entre les fonctions pour représenter les dépendances de contrôle interprocéduraux. i, 238, 263

**Backward-Bounded DSE** Approche de DSE arrière bornée. La propriété de cette analyse est d'être décidable car bornée. Le nombre de chemins à couvrir est donc naturellement fini. i, 63, 263

**bisimulation** En informatique théorique une bisimulation est le fait d'exécuter deux systèmes concomitamment en effectuant la relation bijective entre les transitions des deux systèmes. Par exemple bisimuler deux émulateurs permet à chaque exécution d'instruction de comparer les registres et la mémoire, pour y déceler des différences. i, 52

**bloom-filter** Un bloom filter (filtre de Bloom) est une structure de données probabiliste. Elle permet de s'assurer avec certitude qu'un élément est absent de l'ensemble (pas de faux négatifs) et qu'un élément est présent dans l'ensemble (avec une certaine probabilité, pouvant entraîner des faux-positifs). i, 156

**Breath-First-Search** Stratégie de recherche en largeur dans un arbre itérant l'ensemble des successeurs des nœud du niveau  $n$  avant de répéter l'opération avec les successeurs des nœuds  $n + 1$ . i, 62, 263

**Call Graph** Graphe orienté d'appel de fonctions. Chaque fonction est représentée par un nœud et les appels d'une fonction à l'autre par une arête. i, 63, 263

**Common Weakness Enumeration** Nommage standard géré par le NIST pour catégoriser des faiblesses et des mauvaises pratiques que l'on peut rencontrer dans un programme. i, 23, 263

**Control Flow Graph** Le graphe de flot de contrôle représente tous les chemins pouvant être exécutés par le programme. Les noeuds sont généralement les basic-block et les arrêtes les transitions entre ceux-ci par des sauts conditionnels ou des sauts dynamiques. [i](#), [63](#), [263](#)

**Control Flow Integrity** Technique de vérification de l'intégrité des pointeurs utilisés pour le flot de contrôle du programme. Les pointeurs concerné sont généralement les sauts et les appels (`call`) dynamiques. [i](#), [140](#), [263](#)

**Copy-on-Write** Mécanisme d'optimisation de copie d'objets en mémoire effectuant la copie physiquement que lorsque celle-ci est modifiée. [i](#), [95](#), [263](#)

**Data Flow Graph** Réprésente le graphe de dépendance entre les données d'un programme. Les arcs du graphe représentent généralement les dépendances entre les données par application d'opérations de traitement (arithmétique ou autre). [i](#), [140](#), [263](#)

**Depth-First-Search** Stratégie de recherche en profondeur partant d'un noeud (racine ou non) et explorant l'ensemble des chemins en privilégiant les plus longs/profonds en premier. [i](#), [62](#), [263](#)

**Directed Acyclic Graph** Graphe dirigé et sans cycle. Ce dernier point garanti une complexité très faible pour la plupart des algorithmes de recherche. [i](#), [235](#), [263](#)

**Dynamic Binary Instrumentation** Identique à [DBT](#), un moteur de DBI permet d'instrumenter un programme de manière profonde en exécutant le code d'instrumentation dans le même espace mémoire que le programme. [i](#), [112](#), [263](#)

**Dynamic Symbolic Execution** Aussi appelé exécution concolique est une abréviation utilisée pour désigner le mélange entre l'exécution symbolique et l'exécution concrète. [i](#), [45](#), [263](#)

**Dynamic-Binary-Translation** La translation dynamique de programme est une technique d'instrumentation beaucoup plus invasive mais polyvalente que du debuggin. Cette technique désassemble chaque basic-block les modifies si nécessaire puis les JIT pour les exécuter. [i](#), [209](#), [263](#)

**Ethereum Virtual Machine** Machine virtuelle permettant l'exécution du *bytecode* des *smart-contracts* d'Ethereum. [i](#), [86](#), [264](#)

**Evolutionary Algorithm** Algorithme génétique utilisé pour la mutation des entrées en fuzzing et la sélection des configurations prometteuses. [i](#), [134](#), [263](#)

**Exploit DB** Mécanisme de nommage et de numérotation des vulnérabilités publiées sur le site <https://www.exploit-db.com>. [i](#), [22](#), [264](#)

**Extensible Markup Language** Format textuel, basé sur un mécanisme de balises permettant une structuration de l'information aisée. [i](#), [135](#), [265](#)

**Finite State Machine** Une machine à état contenant un nombre d'états (noeuds) fini. [i](#), [63](#), [264](#)

**Fuzzing Configuration Scheduling** Terme désignant le problème de sélection de configuration dans un algorithme de fuzzing. [i](#), [133](#), [264](#)

**Intermediate Representation** Représentation intermédiaire du code (source ou binaire) utilisée pour effectuer les analyses. Le but étant de manipuler une représentation agnostique du langage source ou de l'architecture cible pour implémenter des analyses portables. [i](#), [48](#), [264](#)

**Just-In-Time** La compilation à la volée désigne la technique consistant à compiler/assembler les instructions ou le bytecode juste avant que celui-ci ne soit exécuté. [i](#), [55](#), [264](#)

**Long Short-Term Memory** Composant d'un réseau RNN permettant de réduire la perte de gradient induite par un réseau RNN. Fonctionnant à base de "porte", elle permet ou non de propager des événements. [i](#), [136](#), [264](#)

**magic-number** Un magic-number (ou magic bytes), est un groupement de plusieurs octets dont la valeur est constante. Ceux-ci permettent par exemple d'identifier un type de fichier. [i](#), [139](#)

**Message Queuing Telemetry Transport** Protocole TCP/IP d'échange de message basé sur le modèle du *publish-subscribe*. Ce protocole est beaucoup utilisé dans les applications de messageries, l'embarqué automobile et de nombreuses applications mobiles. [i](#), [164](#), [264](#)

**mémoïsation** La mémoïsation est une technique d'optimisation de code. Son but est de diminuer le temps d'exécution d'un programme informatique en mémorisant les valeurs retournées par une fonction.. [i](#), [69](#)

**n-grams** Les n-grams sont des séries d'octets de taille n. [i](#), [175](#)

**PHP Hypertext Preprocessor** Langage de scripts orienté serveur, spécialement conçu pour le développement d'applications web. [i](#), [135](#), [264](#)

**Points-To Analysis** Analyse visant à déterminer pour chaque pointeur l'ensemble des valeurs possibles (donc l'ensemble des cibles possibles). [i](#), [249](#), [264](#)

**Process IDentifier** Identifiant unique d'un processus en exécution sur un système. [i](#), [119](#), [264](#)

**Program Dependency Graph** Le graphe de dépendance d'un programme est un graphe utilisé en slicing pour représenter les dépendances de contrôle et les dépendances de données. [i](#), [232](#), [264](#)

**Proof Of Vulnerability** Concept introduit pour le CGC. Un POV représente l'ensemble des entrées qui permettent d'activer un bug afin d'en prouver l'existence. [i](#), [19](#), [264](#)

**Recurrent Neural Network** Réseau de neurones récurrent. Ce type de réseau possède au moins un cycle dans leur structure. Ils sont particulièrement utilisés pour la reconnaissance de parole ou de forme. [i](#), [136](#), [264](#)

**Software Assurance Reference Dataset Project** Projet du NIST regroupant différentes bases de tests pour différents langages visant notamment à tester différentes vulnérabilités (CVE) ou faiblesses (CWE). [i](#), [18](#), [264](#)

**Static Single Assignment** Forme normale de représentation des instructions assurant qu'un même label de variable ne sera assigné qu'une seule fois. Dans la chaîne def-use une variable est donc définie qu'une seule fois. [i](#), [52](#), [264](#)

**System Dependency Graph** Extension des PDG pour représenter les dépendances de manière interprocédurale. [i](#), [232](#), [264](#)

**Value Set Analysis** Analyse de valeurs (usuellement par interprétation abstraite) visant à calculer un ensemble de valeurs possible pour une variable. La taille de l'ensemble est généralement fixe. [i](#), [58](#), [264](#)

**Weakest-Precondition Calculus** Approche orientée par les buts basée sur une logique de Hoare dont le but est de calculer la plus faible précondition satisfaisant une propriété. [i](#), [10](#), [264](#)

**Worst-Case-Execution-Time** Domaine de recherche visant à déterminer pour un programme, un système ou un algorithme le pire cas en termes de temps d'exécution. [i](#), [239](#), [264](#)

---

## Bibliographie

---

- [1] V. 35. *Binary Ninja IL Guide : LLIL*. <https://docs.binary.ninja/dev/bnillil/index.html>. 2018 (cf. p. 50).
- [2] M. ABADI, M. BUDIU, Ú. ERLINGSSON et J. LIGATTI. « Control-flow Integrity Principles, Implementations, and Applications ». In : *ACM Trans. Inf. Syst. Secur.* 13.1 (nov. 2009), 4 :1-4 :40 (cf. p. 140).
- [3] H. J. ABDELNUR, R. STATE et O. FESTOR. « KiF : A Stateful SIP Fuzzer ». In : *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*. IPTComm '07. New York City, New York : ACM, 2007, p. 47-56 (cf. p. 135).
- [4] P. A. ABDULLA, M. FAOUZI ATIG, Y. CHEN, B. P. DIEP, L. HOLÍK, A. REZINE et P. RÜMMER. « Trau : SMT solver for string constraints ». In : *2018 Formal Methods in Computer Aided Design (FMCAD)*. Oct. 2018, p. 1-5 (cf. p. 68).
- [5] H. AGRAWAL et J. R. HORGAN. « Dynamic Program Slicing ». In : *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, White Plains, New York, USA, June 20-22, 1990. 1990, p. 246-256 (cf. p. 238).
- [6] J. AHN, J. AHN, T. HAN et T. HAN. *Static Slicing of a First-Order Functional Language based on Operational Semantics*. Rapp. tech. Operational Semantics, Korea Advanced Institute of Science Technology (KAIST, 1999 (cf. p. 233).
- [7] S. ANAND, P. GODEFROID et N. TILLMANN. « Demand-Driven Compositional Symbolic Execution ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, p. 367-381 (cf. p. 64, 189, 194).
- [8] J. H. ANDREWS, S. HALDAR, Y. LEI et F. C. H. LI. « Tool support for randomized unit testing ». In : *Proceedings of the 1st International Workshop on Random Testing, RT 2006, Portland, Maine, USA, July 20, 2006*. 2006, p. 36-45 (cf. p. 9).
- [9] D. ANDRIESSE. *Practical Binary Analysis*. T. 1. 1. <https://practicalbinaryanalysis.com>. no starch press, déc. 2018 (cf. p. 108).

## BIBLIOGRAPHIE

---

- [10] A. ARMSTRONG et al. « ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS ». In : *PACMPL 3.POPL* (2019), 71 :1-71 :31 (cf. p. 49, 50, 60, 309).
- [11] S. ASADI, M. BLICHA, G. FEDYUKOVICH, A. E. J. HYVÄRINEN, K. EVEN-MENDOZA, N. SHARYGINA et H. CHOCKLER. « Function Summarization Modulo Theories ». In : *22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR*. EasyChair Publications. Ethiopia : EasyChair Publications, 2018 (cf. p. 69).
- [12] C. ASCHERMANN, T. FRASSETTO, T. HOLZ, P. JAUERNIG, A. SADEGHI et D. TEUCHERT. « NAUTILUS : Fishing for Deep Bugs with Grammars ». In : *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. [PDF] [slide] [code] [video]. 2019 (cf. p. 135, 149).
- [13] C. ASCHERMANN, S. SCHUMILO, T. BLAZYTKO, R. GAWLIK et T. HOLZ. « RED-QUEEN : Fuzzing with Input-to-State Correspondence ». In : *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. [PDF]. 2019 (cf. p. 192, 195, 308).
- [14] J.-P. AUMASSON et Y. ROMAILLER. *Automated Testing of Crypto Software Using Differential Fuzzing*. [slides] [code]. 2017 (cf. p. 141).
- [15] T. AVGERINOS, S. K. CHA, B. L. T. HAO et D. BRUMLEY. « AEG : Automatic Exploit Generation ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. [PDF]. 2011 (cf. p. 45, 53, 60, 63, 68, 73).
- [16] T. AVGERINOS, A. REBERT, S. K. CHA et D. BRUMLEY. « Enhancing Symbolic Execution with Veritesting ». en. In : ACM Press, 2014, p. 1083-1094 (cf. p. 45, 65, 103, 195).
- [17] G. BALAKRISHNAN, R. GRUIAN, T. W. REPS et T. TEITELBAUM. « CodeSurfer/x86-A Platform for Analyzing x86 Executables ». In : *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 2005, p. 250-254 (cf. p. 245, 246).
- [18] G. BALAKRISHNAN et T. W. REPS. « Analyzing Memory Accesses in x86 Executables ». In : *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. [PDF]. 2004, p. 5-23 (cf. p. 58, 83, 246).
- [19] R. BALDONI, E. COPPA, D. C. D'ELIA, C. DEMETRESCU et I. FINOCCHI. « A Survey of Symbolic Execution Techniques ». In : *ACM Comput. Surv.* 51.3 (2018), 50 :1-50 :39 (cf. p. 9, 46, 57).
- [20] M. BANKOVIC. « Extending SMT solvers with support for finite domain alldifferent constraint ». In : *Constraints* 21.4 (2016), p. 463-494 (cf. p. 312).

- [21] G. BANKS, M. COVA, V. FELMETSGER, K. ALMEROOTH, R. KEMMERER et G. VIGNA. « SNOOZE : Toward a Stateful NetwOrk prOtocol fuzzEr ». In : *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, p. 343-358 (cf. p. 135).
- [22] S. BARDIN, R. DAVID et J. MARION. « Backward-Bounded DSE : Targeting Infeasibility Questions on Obfuscated Codes ». In : *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 2017, p. 633-651 (cf. p. 63, 64, 240).
- [23] C. BARRET, P. FONTAINE et C. TINELLI. *The SMT-LIB Standard Version 2.6*. Rapp. tech. [PDF]. Juill. 2017 (cf. p. 67).
- [24] J. B. BARROS, D. C. da CRUZ, P. R. HENRIQUES et J. S. PINTO. « Assertion-based slicing and slice graphs ». In : *Formal Asp. Comput.* 24.2 (2012), p. 217-248 (cf. p. 239).
- [25] O. BASTANI, R. SHARMA, A. AIKEN et P. LIANG. « Synthesizing Program Input Grammars ». In : *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. [PDF] [video]. Barcelona, Spain : ACM, 2017, p. 95-110 (cf. p. 136, 149).
- [26] F. BELLARD. « QEMU, a Fast and Portable Dynamic Translator ». In : *Proceedings of the FREENIX Track : 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. 2005, p. 41-46 (cf. p. 23, 49, 50, 105, 132, 133, 156, 201, 220).
- [27] J. BERGERETTI et B. CARRÉ. « Information-Flow and Data-Flow Analysis of while-Programs ». In : *ACM Trans. Program. Lang. Syst.* 7.1 (1985), p. 37-61 (cf. p. 232).
- [28] Á. BESZÉDES, C. FARAGÓ, Z. M. SZABÓ, J. CSIRIK et T. GYIMÓTHY. « Union Slices for Program Maintenance ». In : *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*. 2002, p. 12-21 (cf. p. 239).
- [29] Á. BESZÉDES, T. GERGELY, Z. M. SZABÓ, J. CSIRIK et T. GYIMÓTHY. « Dynamic Slicing Method for Maintenance of Large C Programs ». In : *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*. 2001, p. 105-113 (cf. p. 238).
- [30] B. BEURDOUCHE et al. « A Messy State of the Union : Taming the Composite State Machines of TLS ». In : *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, p. 535-552 (cf. p. 146).
- [31] S. BHANSALI et al. « Framework for Instruction-level Tracing and Analysis of Program Executions ». In : *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada : ACM, 2006, p. 154-163 (cf. p. 133).

## BIBLIOGRAPHIE

---

- [32] D. W. BINKLEY, N. GOLD, M. HARMAN, S. S. ISLAM, J. KRINKE et S. YOO. « ORBS : language-independent program slicing ». In : *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 2014, p. 109-120 (cf. p. 239).
- [33] T. of BITS. *Manticore : Symbolic execution for humans.* <https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans>. 2017 (cf. p. 11, 73, 77, 115, 208).
- [34] T. of BITS. *Sienna Locomotive.* [\[code\]](#). 2019 (cf. p. 149).
- [35] T. of BITS. *GRR* (cf. p. 138, 149).
- [36] T. of BITS. *McSema.* [\[code\]](#) (cf. p. 48, 51, 215).
- [37] T. of BITS. *Remill.* [\[code\]](#) (cf. p. 51).
- [38] P. E. BLACK. « Software Assurance Metrics and Tool Evaluation ». In : *Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2005, Las Vegas, Nevada, USA, June 27-29, 2005, Volume 2.* 2005, p. 829-835 (cf. p. 18).
- [39] M. BÖHME, V. PHAM et A. ROYCHOUDHURY. « Coverage-based Greybox Fuzzing as Markov Chain ». In : *IEEE Transactions on Software Engineering* (2018). [\[PDF\]](#) [\[video\]](#), p. 1-1 (cf. p. 134, 149, 220).
- [40] M. BÖHME, V. PHAM, M. NGUYEN et A. ROYCHOUDHURY. « Directed Greybox Fuzzing ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* [\[PDF\]](#) [\[video\]](#). 2017, p. 2329-2344 (cf. p. 134, 143, 149).
- [41] B. BOISSINOT, A. DARTE, F. RASTELLO, B. D. de DINECHIN et C. GUILLOU. « Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency ». In : *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009.* 2009, p. 114-125 (cf. p. 52).
- [42] T. BOLAND et P. E. BLACK. « Juliet 1.1 C/C++ and Java Test Suite ». In : *IEEE Computer* 45.10 (2012), p. 88-90 (cf. p. 18).
- [43] P. BOONSTOPPEL, C. CADAR et D. R. ENGLER. « RWset : Attacking Path Explosion in Constraint-Based Test Generation ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 2008, p. 351-366 (cf. p. 65).
- [44] E. BOUNIMOVA, P. GODEFROID et D. A. MOLNAR. « Billions and billions of constraints : whitebox fuzz testing in production ». In : *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* [\[PDF\]](#). 2013, p. 122-131 (cf. p. 190).

- [45] T. BOUTON, D. C. B. D. OLIVEIRA, D. DÉHARBE et P. FONTAINE. « veriT : An Open, Trustable and Efficient SMT-Solver ». In : *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings.* 2009, p. 151-156 (cf. p. 69).
- [46] G. BRAT, J. A. NAVAS, N. SHI et A. VENET. « IKOS : A Framework for Static Analysis Based on Abstract Interpretation ». In : *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings.* [PDF] [code]. 2014, p. 271-277 (cf. p. 256).
- [47] R. K. BRAYTON et A. MISHCHENKO. « ABC : An Academic Industrial-Strength Verification Tool ». In : *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings.* 2010, p. 24-40 (cf. p. 69, 105).
- [48] D. BRUENING, Q. ZHAO et S. P. AMARASINGHE. « Transparent dynamic instrumentation ». In : *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012).* 2012, p. 133-144 (cf. p. 105, 132, 133).
- [49] D. BRUMLEY, I. JAGER, T. AVGERINOS et E. J. SCHWARTZ. « BAP : A Binary Analysis Platform ». In : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* [code]. 2011, p. 463-469 (cf. p. 49, 50, 73, 240).
- [50] R. BRUMMAYER et A. BIERE. « Boolector : An Efficient SMT Solver for Bit-Vectors and Arrays ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* 2009, p. 174-177 (cf. p. 69, 96).
- [51] S. BUCUR. « Improving Scalability of Symbolic Execution for Software with Complex Environment Interfaces ». In : (2015). [PDF], p. 124 (cf. p. 60).
- [52] S. BUCUR, V. URECHE, C. ZAMFIR et G. CANDEA. « Parallel symbolic execution for automated real-world software testing ». In : *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011.* [site]. 2011, p. 183-198 (cf. p. 60, 73).
- [53] J. BURNIM et K. SEN. « Heuristics for Scalable Dynamic Test Generation ». In : *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy.* [PDF] [page] [code]. 2008, p. 443-446 (cf. p. 73).
- [54] J. CABALLERO, P. POOSANKAM, S. MCCAMANT, D. BABIC et D. SONG. « Input generation via decomposition and re-stitching : finding bugs in Malware ». In : *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010.* [PDF]. 2010, p. 413-425 (cf. p. 191, 192, 260).

## BIBLIOGRAPHIE

---

- [55] J. CABALLERO, H. YIN, Z. LIANG et D. SONG. « Polyglot : Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis ». In : *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA : ACM, 2007, p. 317-329 (cf. p. 136).
- [56] C. CADAR, D. DUNBAR et D. R. ENGLER. « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In : *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. [PDF] [site]. 2008, p. 209-224 (cf. p. 53, 58, 62, 68, 69, 73, 77, 94, 220, 240, 248).
- [57] C. CADAR, V. GANESH, P. M. PAWLOWSKI, D. L. DILL et D. R. ENGLER. « EXE : Automatically Generating Inputs of Death ». In : *ACM Trans. Inf. Syst. Secur.* 12.2 (2008). [PDF], 10 :1-10 :38 (cf. p. 58, 60, 68, 69, 73).
- [58] C. CADAR, P. GODEFROID, S. KHURSHID, C. S. PĂSĂREANU, K. SEN, N. TILLMANN et W. VISSER. « Symbolic Execution for Software Testing in Practice : Preliminary Assessment ». In : ACM Press, 2011, p. 1066 (cf. p. 45).
- [59] C. CADAR et K. SEN. « Symbolic Execution for Software Testing : Three Decades Later ». In : *Communications of the ACM* 56.2 (fév. 2013), p. 82 (cf. p. 45).
- [60] Y. CAI et W. K. CHAN. « MagicFuzzer : Scalable deadlock detection for large-scale applications ». In : *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. [PDF] [slides]. 2012, p. 606-616 (cf. p. 149).
- [61] G. CAMPANA. *Fuzzgrind : un outil de fuzzing automatique*. [PDF]. 2009 (cf. p. 192).
- [62] G. CANFORA, A. CIMITILE et A. D. LUCIA. « Conditioned program slicing ». In : *Information & Software Technology* 40.11-12 (1998), p. 595-607 (cf. p. 238).
- [63] S. K. CHA, M. WOO et D. BRUMLEY. « Program-Adaptive Mutational Fuzzing ». In : *2015 IEEE Symposium on Security and Privacy*. [PDF]. Mai 2015, p. 725-741 (cf. p. 137, 149, 190, 191).
- [64] S. K. CHA, T. AVGERINOS, A. REBERT et D. BRUMLEY. « Unleashing Mayhem on Binary Code ». In : *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. [PDF]. 2012, p. 380-394 (cf. p. 53, 56, 58, 63, 68, 73, 77, 104, 256, 258, 259).
- [65] S. CHA, S. HONG, J. LEE et H. OH. « Automatically generating search heuristics for concolic testing ». In : *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. [PDF]. 2018, p. 1244-1254 (cf. p. 62).
- [66] M. CHALUPA. « Slicing of LLVM bitcode ». [thesis]. Mém. de mast. Masaryk University, 2016 (cf. p. 249).
- [67] S. CHANDRA, S. J. FINK et M. SRIDHARAN. « Snufflebug : a powerful approach to weakest preconditions ». In : *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 2009, p. 363-374 (cf. p. 11).

- [68] O. CHEBARO, P. CUOQ, N. KOSMATOV, B. MARRE, A. PACALET, N. WILLIAMS et B. YAKOBOWSKI. « Behind the Scenes in SANTE : A Combination of Static and Dynamic Analyses ». In : *Automated Software Engg.* 21.1 (mars 2014), p. 107-143 (cf. p. 239).
- [69] H. CHEN, Y. XUE, Y. LI, B. CHEN, X. XIE, X. WU et Y. LIU. « Hawkeye : Towards a Desired Directed Grey-box Fuzzer ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. [PDF] [site]. 2018, p. 2095-2108 (cf. p. 144, 149).
- [70] J. CHEN et al. « IoTFuzzer : Discovering Memory Corruptions in IoT Through App-based Fuzzing ». In : *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. [PDF] [slides] [video]. 2018 (cf. p. 149).
- [71] P. CHEN et H. CHEN. « Angora : Efficient Fuzzing by Principled Search ». In : *2018 IEEE Symposium on Security and Privacy (SP)* (2018), p. 711-725 (cf. p. 143, 190-192, 195, 202, 217, 218).
- [72] T. CHEN, X. LIN, J. HUANG, A. BACCHUS et X. ZHANG. « An empirical investigation into path divergences for concolic execution using CREST ». In : *Security and Communication Networks* 8.18 (2015), p. 3667-3681 (cf. p. 59).
- [73] T. CHEN, X. ZHANG, S. GUO, H. LI et Y. WU. « State of the art : Dynamic symbolic execution for automated test generation ». In : *Future Generation Comp. Syst.* 29.7 (2013), p. 1758-1773 (cf. p. 46).
- [74] Y. CHEN, A. GROCE, C. ZHANG, W.-K. WONG, X. FERN, E. EIDE et J. REGEHR. « Taming Compiler Fuzzers ». In : *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13*. Seattle, Washington, USA : ACM, 2013, p. 197-208 (cf. p. 142).
- [75] Y. CHEN, T. SU, C. SUN, Z. SU et J. ZHAO. « Coverage-directed Differential Testing of JVM Implementations ». In : *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16*. Santa Barbara, CA, USA : ACM, 2016, p. 85-99 (cf. p. 141).
- [76] Y. CHEN, T. SU, C. SUN, Z. SU et J. ZHAO. « Coverage-directed differential testing of JVM implementations ». In : *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. [PDF] [video]. 2016, p. 85-99 (cf. p. 149).
- [77] V. CHIPOUNOV, V. KUZNETSOV et G. CANDEA. « S2E : a platform for in-vivo multi-path analysis of software systems ». In : *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. [PDF]. 2011, p. 265-278 (cf. p. 51, 53-55, 69, 73).
- [78] J. CHOI, J. JANG, C. HAN et S. K. CHA. « Grey-box Concolic Testing on Binary Code ». In : *Proceedings of 41th International Conference on Software Engineering*. [PDF]. 2019 (cf. p. 192, 195).

## BIBLIOGRAPHIE

---

- [79] A. B. CHOWDHURY, R. K. MEDICHERLA et R. VENKATESH. « VeriFuzz : Program Aware Fuzzing - (Competition Contribution) ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS : TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. 2019, p. 244-249 (cf. p. 149).
- [80] J. CHRIST, J. HOENICKE et A. NUTZ. « SMTInterpol : An Interpolating SMT Solver ». In : *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. 2012, p. 248-254 (cf. p. 69).
- [81] CIFASIS. *Neural-Fuzzer* (cf. p. 136).
- [82] A. CIMATTI, A. GRIGGIO, B. J. SCHAAFSMA et R. SEBASTIANI. « The MathSAT5 SMT Solver ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, p. 93-107 (cf. p. 69).
- [83] COINBASE. *Maxfuzz - Containerized Cloud Fuzzing* (cf. p. 150).
- [84] C. COLLBERG, S. MARTIN, J. MYERS et J. NAGRA. « Distributed Application Tamper Detection via Continuous Software Updates ». In : *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida : ACM, 2012, p. 319-328 (cf. p. 48).
- [85] P. M. COMPARETTI, G. WONDRACEK, C. KRUEGEL et E. KIRDA. « Prospex : Protocol Specification Extraction ». In : *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. SP '09. Washington, DC, USA : IEEE Computer Society, 2009, p. 110-125 (cf. p. 136).
- [86] J. J. COMUZZI et J. M. HART. « Program Slicing Using Weakest Preconditions ». In : *FME '96 : Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*. 1996, p. 557-575 (cf. p. 239).
- [87] E. COPPA, D. C. D'ELIA et C. DEMETRESCU. « Rethinking pointer reasoning in symbolic execution ». In : *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, p. 613-618 (cf. p. 58).
- [88] J. CORINA, A. MACHIRY, C. SALLS, Y. SHOSHITAISHVILI, S. HAO, C. KRUEGEL et G. VIGNA. « DIFUZE : Interface Aware Fuzzing for Kernel Drivers ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. [PDF] [code]. 2017, p. 2123-2138 (cf. p. 149).
- [89] M. CORPORATION. *!exploitable Crash Analyzer – MSEC Debugger Extensions* (cf. p. 142).

- [90] F. CORZILIUS, G. KREMER, S. JUNGES, S. SCHUPP et E. ÁBRAHÁM. « SMT-RAT : An Open Source C++ Toolbox for Strategic and Parallel SMT Solving ». In : *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*. 2015, p. 360-368 (cf. p. 69).
- [91] M. COSTA, M. CASTRO, L. ZHOU, L. ZHANG et M. PEINADO. « Bouncer : securing software by blocking bad input ». In : *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. [PDF]. 2007, p. 117-130 (cf. p. 73).
- [92] L. CSEPENTO et Z. MICSKEI. « Evaluating Symbolic Execution-Based Test Tools ». In : *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 2015, p. 1-10 (cf. p. 46).
- [93] W. CUI, M. PEINADO, S. K. CHA, Y. FRATANTONIO et V. P. KEMERLIS. « RE-Tracer : Triaging Crashes by Reverse Execution from Partial Memory Dumps ». In : *Proceedings of the 38th International Conference on Software Engineering. ICSE '16*. Austin, Texas : ACM, 2016, p. 820-831 (cf. p. 142).
- [94] W. CUI, M. PEINADO, K. CHEN, H. J. WANG et L. IRUN-BRIZ. « Tupni : Automatic Reverse Engineering of Input Formats ». In : *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA : ACM, 2008, p. 391-402 (cf. p. 136).
- [95] S. DASGUPTA, D. PARK, T. KASAMPALIS, V. S. ADVE et G. ROŞU. « A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture ». In : *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. [PDF] [code]. ACM, juin 2019 (cf. p. 49).
- [96] *DataFlowSanitizer*. <https://clang.llvm.org/docs/DataFlowSanitizer.html> (cf. p. 140).
- [97] R. DAVID. « Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes. (Approches formelles de désobfuscation automatique et de rétro-ingénierie de codes protégés) ». Thèse de doct. University of Lorraine, Nancy, France, 2017 (cf. p. 70).
- [98] R. DAVID, S. BARDIN, J. FEIST, L. MOUNIER, M. POTET, T. D. TA et J. MARION. « Specification of concretization and symbolization policies in symbolic execution ». In : *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 2016, p. 36-46 (cf. p. 54, 58).
- [99] R. DAVID, S. BARDIN, T. D. TA, L. MOUNIER, J. FEIST, M.-L. POTET et J.-Y. MARION. « BINSEC/SE : A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis ». In : *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*. 2016, p. 653-656 (cf. p. 53, 58, 73, 241).
- [100] J. D. DEMOTT, R. J. ENBODY et W. F. *Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing*. [PDF]. 2007 (cf. p. 137).

## BIBLIOGRAPHIE

---

- [101] L. DEROSE, J. BROWN, C. FIGUEIRA et R. WISMÜLLER. *DynInst : Putting the Performance in High Performance Computing*. <https://www.dyninst.org/dyninst> (cf. p. 132, 133).
- [102] M. DETERS, A. REYNOLDS, T. KING, C. W. BARRETT et C. TINELLI. « A Tour of CVC4 : How It Works, and How to Use It ». In : *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. 2014, p. 7 (cf. p. 69, 105).
- [103] W. DIETZ, P. LI, J. REGEHR et V. ADVE. « Understanding Integer Overflow in C/C++ ». In : *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. [PDF]. Zurich, Switzerland : IEEE Press, 2012, p. 760-770 (cf. p. 141, 168).
- [104] A. DINABURD. *Making a Scalable Automated Hacking System, From DevOps to Pwning*. [slides]. 2016 (cf. p. 213).
- [105] A. DINABURG et P. CUOQ. *Zlib Automated Security Assessment*. Rapp. tech. [report]. 2016, p. 24 (cf. p. 216).
- [106] P. DINGES et G. A. AGHA. « Targeted test input generation using symbolic-concrete backward execution ». In : *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Västerås, Sweden - September 15 - 19, 2014*. 2014, p. 31-36 (cf. p. 63).
- [107] P. DINGES et G. A. AGHA. « Targeted test input generation using symbolic-concrete backward execution ». In : *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Västerås, Sweden - September 15 - 19, 2014*. 2014, p. 31-36 (cf. p. 64).
- [108] A. DJOUDI, S. BARDIN et É. GOUBAULT. « Recovering High-Level Conditions from Binary Programs ». In : *FM 2016 : Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. 2016, p. 235-253 (cf. p. 50, 52).
- [109] B. DOLAN-GAVITT et al. « LAVA : Large-Scale Automated Vulnerability Addiction ». In : *2016 IEEE Symposium on Security and Privacy (SP)*. Mai 2016, p. 110-121 (cf. p. 22, 208, 307).
- [110] B. DOLAN-GAVITT, J. HODOSH, P. HULIN, T. LEEK et R. WHELAN. « Repeatable Reverse Engineering with PANDA ». In : *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*. 2015, 4 :1-4 :11 (cf. p. 23, 105).
- [111] A. DOUPÉ, L. CAVEDON, C. KRUEGEL et G. VIGNA. « Enemy of the State : A State-Aware Black-Box Web Vulnerability Scanner ». In : *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 2012, p. 523-538 (cf. p. 147).
- [112] T. DULLIEN et S. PORST. « REIL : A platform-independent intermediate representation of disassembled code for static code analysis ». In : [PDF]. 2009 (cf. p. 48-50).

- [113] D. DURAN, D. WESTON et M. MILLER. *Targeted Taint Driven Fuzzing using Software Metrics*. 2011 (cf. p. 191).
- [114] B. DUTERTRE. « Yices 2.2 ». In : *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, p. 737-744 (cf. p. 69).
- [115] M. EDDINGTON. *Peach Fuzzing Platform* (cf. p. 135, 143, 145, 149).
- [116] A *QBDI-based Fuzzer Taming Magic Bytes*. [PDF]. CEUR Workshop Proceedings, 2019 (cf. p. 36, 37, 184).
- [117] D. R. ENGLER et D. DUNBAR. « Under-constrained execution : making automatic code destruction easy and scalable ». In : *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. 2007, p. 1-4 (cf. p. 195, 209).
- [118] ERNW. *Dizzy : Network and USB protocol fuzzing toolkit*. [code] (cf. p. 149).
- [119] B. FARINIER, R. DAVID, S. BARDIN et M. LEMERRE. « Arrays Made Simpler : An Efficient, Scalable and Thorough Preprocessing ». In : *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. 2018, p. 363-380 (cf. p. 59, 70, 90).
- [120] J. FEIST, L. MOUNIER, S. BARDIN, R. DAVID et M. POTET. « Finding the needle in the heap : combining static analysis and dynamic symbolic execution to trigger use-after-free ». In : *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016, Los Angeles, California, USA, December 5-6, 2016*. [PDF]. 2016, 2 :1-2 :12 (cf. p. 233, 240, 243).
- [121] J. FERRANTE, K. J. OTTENSTEIN et J. D. WARREN. « The Program Dependence Graph and Its Use in Optimization ». In : *ACM Trans. Program. Lang. Syst.* 9.3 (1987), p. 319-349 (cf. p. 235, 250).
- [122] F. FKIE. *Maxfuzz - Containerized Cloud Fuzzing* (cf. p. 150).
- [123] E. FLEURY, O. LY, G. POINT et A. VINCENT. « Insight : An Open Binary Analysis Framework ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, p. 218-224 (cf. p. 50).
- [124] J. FOOTE. *GDB exploitable*. <https://github.com/jfoote/exploitable> (cf. p. 142).
- [125] I. FORGACS et T. GYIMOTHY. *An Efficient Interprocedural Slicing Method for Large Programs*. Rapp. tech. 1997 (cf. p. 236).
- [126] C. FOX, S. DANICIC, M. HARMAN et R. M. HIERONS. « ConSIT : a fully automated conditioned program slicer ». In : *Softw., Pract. Exper.* 34.1 (2004), p. 15-46 (cf. p. 239).
- [127] J. GALEA, S. HEELAN, D. NEVILLE et D. KROENING. « Evaluating Manual Intervention to Address the Challenges of Bug Finding with KLEE ». In : *CoRR* abs/1805.03450 (2018). [PDF] (cf. p. 62, 71, 95).

## BIBLIOGRAPHIE

---

- [128] S. GAN, C. ZHANG, X. QIN, X. TU, K. LI, Z. PEI et Z. CHEN. « CollAFL : Path Sensitive Fuzzing ». In : *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, mai 2018 (cf. p. 144, 149).
- [129] V. GANESH. *STP Constraint Solver*. [stp.github.io](https://stp.github.io). 2016 (cf. p. 68, 69, 96).
- [130] V. GANESH et D. L. DILL. « A Decision Procedure for Bit-Vectors and Arrays ». In : *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, p. 519-531 (cf. p. 56, 68).
- [131] V. GANESH, T. LEEK et M. C. RINARD. « Taint-based directed whitebox fuzzing ». In : *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. [\[PDF\]](#). 2009, p. 474-484 (cf. p. 190, 192).
- [132] H. GASCON, C. WRESSNEGGER, F. YAMAGUCHI, D. ARP et K. RIECK. « Pulsar : Stateful Black-Box Fuzzing of Proprietary Network Protocols ». In : *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. [\[PDF\]](#) [\[code\]](#). Springer International Publishing, 2015, p. 330-347 (cf. p. 130, 136, 149, 174, 177).
- [133] H. GASCON, C. WRESSNEGGER, F. YAMAGUCHI, D. ARP et K. RIECK. *PULSAR - Protocol Learning and Stateful Fuzzing* (cf. p. 174).
- [134] J. GELDENHUYS, M. B. DWYER et W. VISSER. « Probabilistic Symbolic Execution ». In : *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA : ACM, 2012, p. 166-176 (cf. p. 61).
- [135] P. GODEFROID. « Compositional dynamic test generation ». In : *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 2007, p. 47-54 (cf. p. 189, 194).
- [136] P. GODEFROID. « Random testing for security : blackbox vs. whitebox fuzzing ». In : *In RT '07 : Proceedings of the 2nd international workshop on Random testing*. ACM, 2007 (cf. p. 130).
- [137] P. GODEFROID, N. KLARLUND et K. SEN. « DART : directed automated random testing ». In : *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. [\[PDF\]](#). 2005, p. 213-223 (cf. p. 53, 58, 60, 62, 73).
- [138] P. GODEFROID, M. Y. LEVIN et D. A. MOLNAR. « Automated Whitebox Fuzz Testing ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. [\[PDF\]](#). 2008 (cf. p. 82, 189).
- [139] P. GODEFROID, M. Y. LEVIN et D. A. MOLNAR. « SAGE : whitebox fuzzing for security testing ». In : *Commun. ACM* 55.3 (2012). [\[PDF\]](#), p. 40-44 (cf. p. 45, 53, 58, 59, 62, 68, 73, 189, 190, 192).

- [140] P. GODEFROID et D. LUCHAUP. « Automatic partial loop summarization in dynamic test generation ». In : *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011.* 2011, p. 23-33 (cf. p. 65).
- [141] P. GODEFROID, H. PELEG et R. SINGH. « Learn&#38;Fuzz : Machine Learning for Input Fuzzing ». In : *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering.* ASE 2017. Urbana-Champaign, IL, USA : IEEE Press, 2017, p. 50-59 (cf. p. 136).
- [142] P. GOODMAN et A. DINABURG. « The Past, Present, and Future of Cyberdyne ». In : *IEEE Security Privacy* 16.2 (2018), p. 61-69 (cf. p. 144, 193, 213).
- [143] P. GOODMAN. *Cyberdyne : Automatic bug-finding at scale.* [slides]. 2019 (cf. p. 213).
- [144] P. GOODMAN, G. GRIECO et A. GROCE. « Tutorial : DeepState : Bringing Vulnerability Detection Tools into the Development Cycle ». In : *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018.* 2018, p. 130-131 (cf. p. 221).
- [145] GOOGLE. *ClusterFuzz - scalable fuzzing infrastructure.* [code] (cf. p. 148).
- [146] GOOGLE. *OSS-Fuzz - Continuous Fuzzing for Open Source Software.* <https://github.com/google/oss-fuzz> [code] (cf. p. 148, 163).
- [147] D. GOSWAMI et R. MALL. « An efficient method for computing dynamic program slices ». In : *Inf. Process. Lett.* 81.2 (2002), p. 111-117 (cf. p. 238).
- [148] I. GOTOVCHITS, R. V. TONDER et D. BRUMLEY. « Saluki : Finding Taint-style Vulnerabilities with Static Property Checking ». In : [PDF]. 2018 (cf. p. 261).
- [149] GRAMMATECH. *GTIRB : Intermediate Representation for Binary analysis and transformation.* 2019 (cf. p. 51).
- [150] A. GROCE, J. HOLMES, D. MARINOV, A. SHI et L. ZHANG. « An extensible, regular-expression-based tool for multi-language mutant generation ». In : *Proceedings of the 40th International Conference on Software Engineering : Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* [code]. 2018, p. 25-28 (cf. p. 135).
- [151] S. GROSS. *FuzzIL : Guided Fuzzing For Javascript Engines.* [PDF] [code]. 14 fév. 2019 (cf. p. 135).
- [152] N. GROUP. *TriforceAFL : AFL/QEMU fuzzing with full-system emulation.* [code]. 2016 (cf. p. 149).
- [153] N. GROUP. *Triforce Linux Syscall Fuzzer* (cf. p. 136).
- [154] D. GUIDO. *Your tool works better than mine ? Prove it.* [URL] (cf. p. 19).
- [155] I. GUILFANOV. *Decompiler Internals : Microcode.* [PDF]. 2018 (cf. p. 50).

## BIBLIOGRAPHIE

---

- [156] R. GUPTA et M. L. SOFFA. « Hybrid Slicing : An Approach for Refining Static Slices Using Dynamic Information ». In : *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995.* 1995, p. 29-40 (cf. p. 239).
- [157] J. GUTIERREZ, M. ESCALONA, M. MEJÍAS et J. TORRES. « Generation of test cases from functional requirements. A survey ». In : (juill. 2008) (cf. p. 9).
- [158] I. HALLER, Y. JEON, H. PENG, M. PAYER, C. GIUFFRIDA, H. Bos et E. van der KOUWE. « TypeSan : Practical Type Confusion Detection ». In : *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 2016, p. 517-528 (cf. p. 140).
- [159] I. HALLER, Y. JEON, H. PENG, M. PAYER, C. GIUFFRIDA, H. Bos et E. van der KOUWE. « TypeSan : Practical Type Confusion Detection ». In : *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 2016, p. 517-528 (cf. p. 140).
- [160] I. HALLER, A. SLOWINSKA, M. NEUGSCHWANDTNER et H. Bos. « Dowsing for Overflows : A Guided Fuzzer to Find Buffer Boundary Violations ». In : *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013.* [PDF] [slides] [video]. 2013, p. 49-64 (cf. p. 191, 192).
- [161] H. HAN et S. K. CHA. « IMF : Inferred Model-based Fuzzer ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* [PDF] [code] [video]. 2017, p. 2345-2358 (cf. p. 149).
- [162] W. HAN, B. JOE, B. LEE, C. SONG et I. SHIN. « Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing ». In : *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* 2018 (cf. p. 140).
- [163] T. HANSEN. « A constraint solver and its application to machine code test generation ». Thèse de doct. University of Melbourne, Australia, 2012 (cf. p. 69).
- [164] T. HANSEN, P. SCHACHTÉ et H. SØNDERGAARD. « State Joining and Splitting for the Symbolic Execution of Binaries ». In : *Runtime Verification*. Springer Berlin Heidelberg, 2009, p. 76-92 (cf. p. 65).
- [165] M. HARMAN. « Automated Test Data Generation using Search Based Software Engineering ». In : *Proceedings of the Second International Workshop on Automation of Software Test, AST 2007, Minneapolis, MN, USA, May 26-26, 2007.* 2007, p. 1-2 (cf. p. 9).
- [166] M. HARMAN, D. W. BINKLEY et S. DANICIC. « Amorphous program slicing ». In : *Journal of Systems and Software* 68.1 (2003), p. 45-64 (cf. p. 239).
- [167] M. HARMAN, R. M. HIERONS, C. FOX, S. DANICIC et J. HOWROYD. « Pre/Post Conditioned Slicing ». In : *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001.* 2001, p. 138-147 (cf. p. 239).

- [168] M. J. HARROLD et N. CI. « Reuse-Driven Interprocedural Slicing ». In : *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*. 1998, p. 74-83 (cf. p. 238).
- [169] S. HEELAN. « Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities ». [thesis]. Mém. de mast. University of Oxford, 2009 (cf. p. 83).
- [170] S. HEELAN et A. GIANNI. « Augmenting Vulnerability Analysis of Binary Code ». In : *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. New York, NY, USA : ACM, 2012, p. 199-208 (cf. p. 45).
- [171] C. HEITMAN et I. ARCE. *BARF : A Multiplatform Open Source Binary Analysis and Reverse Engineering Framework*. [PDF]. 2014 (cf. p. 49).
- [172] A. HELIN. *Radamsa*. [code] (cf. p. 130, 135, 137, 149, 214).
- [173] M. HIND. « Pointer analysis : haven't we solved this problem yet ? » In : *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*. 2001, p. 54-61 (cf. p. 235).
- [174] S. HOCEVAR. *zzuf*. <https://github.com/samhocevar/zzuf>. 2016 (cf. p. 133, 149).
- [175] J. HOFFMANN et J. KOEHLER. « A New Method to Index and Query Sets ». In : *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*. 1999, p. 462-467 (cf. p. 96).
- [176] C. HOLLER, K. HERZIG et A. ZELLER. « Fuzzing with Code Fragments ». In : *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. [PDF] [slides] [video]. Bellevue, WA : USENIX, 2012, p. 445-458 (cf. p. 135, 149).
- [177] S. HORWITZ, T. REPS et D. BINKLEY. « Interprocedural Slicing Using Dependence Graphs ». In : *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA : ACM, 1988, p. 35-46 (cf. p. 232, 235).
- [178] S. HORWITZ, P. PFEIFFER et T. W. REPS. « Dependence Analysis for Pointer Variables ». In : *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. 1989, p. 28-40 (cf. p. 235).
- [179] A. HOUSEHOLDER. *Well There's Your Problem : Isolating the Crash-Inducing Bits in a Fuzzed File*. Rapp. tech. CMU/SEI-2012-TN-018. Pittsburgh, PA : Software Engineering Institute, Carnegie Mellon University, 2012 (cf. p. 142).
- [180] A. D. HOUSEHOLDER et J. M. FOOTE. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*. 2012 (cf. p. 134).
- [181] C. HUBAIN et C. TESSIER. *Implementing an LLVM based Dynamic Binary Instrumentation Framework*. 2017 (cf. p. 8, 132, 133, 258).

## BIBLIOGRAPHIE

---

- [182] M. HUTCHINS, H. FOSTER, T. GORADIA et T. OSTRAND. « Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria ». In : *Proceedings of 16th International Conference on Software Engineering*. Mai 1994, p. 191-200 (cf. p. 18).
- [183] M. IGUERNELALA. « Strengthening the Heart of an SMT-Solver : Design and Implementation of Efficient Decision Procedures ». Thèse de Doctorat. Université Paris-Sud, juin 2013 (cf. p. 69).
- [184] A. INC. *Accessing CrashWrangler to analyze crashes for security implications*. Rapp. tech. 2014 (cf. p. 142).
- [185] G. INC. *Dependence Graphs and Program Slicing : CodeSurfer Technology Overview*. Rapp. tech. [PDF]. 1999, p. 14 (cf. p. 234, 243, 245).
- [186] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*. [PDF]. Intel Corporation. Août 2007 (cf. p. 111, 164).
- [187] INTEL. *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*. [manuel]. Intel Corporation. 2019 (cf. p. 164).
- [188] V. IOZZO. *0-knowledge fuzzing*. [PDF]. 2010 (cf. p. 191).
- [189] ISO. *ISO/IEC 9899 :2011 Information technology — Programming languages — C*. Geneva, Switzerland : International Organization for Standardization, déc. 2011, 683 (est.) (Cf. p. 39).
- [190] D. JACKSON et E. J. ROLLINS. *Chopping : A Generalization of Slicing*. Rapp. tech. Pittsburgh, PA, USA, 1994 (cf. p. 232).
- [191] J. JAFFAR, V. MURALI, J. A. NAVAS et A. E. SANTOSA. « Path-Sensitive Backward Slicing ». In : *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 2012, p. 231-247 (cf. p. 240).
- [192] W. JOHANSSON, M. SVENSSON, U. E. LARSON, M. ALMGREN et V. GULISANO. « T-Fuzz : Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols ». In : *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014, p. 323-332 (cf. p. 135).
- [193] R. JOHNSON. *Taint Driven Crash Analysis*. [slides]. 2018 (cf. p. 240).
- [194] D. JONES. *Trinity* (cf. p. 135).
- [195] P. JOSHI, C. PARK, K. SEN et M. NAIK. « A randomized dynamic program analysis technique for detecting real deadlocks ». In : *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. [PDF]. 2009, p. 110-120 (cf. p. 149).
- [196] J. JUNG, A. SHETH, B. GREENSTEIN, D. WETHERALL, G. MAGANIS et T. KOHNO. « Privacy Oracle : A System for Finding Application Leaks with Black Box Differential Testing ». In : *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA : ACM, 2008, p. 279-288 (cf. p. 141).

- [197] M. JUNG, S. KIM, H. HAN, J. CHOI et S. K. CHA. « B2R2 : Building an Efficient Front-End for Binary Analysis ». In : *Proceedings of the NDSS Workshop on Binary Analysis Research*. [PDF] [site] [code] [artefact]. 2019 (cf. p. 49, 50, 52).
- [198] R. KAKSONEN, M. LAAKSO et A. TAKANEN. « Software Security Assessment through Specification Mutations and Fault Injection ». In : *Communications and Multimedia Security Issues of the New Century*. Springer US, 2001, p. 173-183 (cf. p. 135).
- [199] M. G. KANG, S. MCCAMANT, P. POOSANKAM et D. SONG. « DTA++ : Dynamic Taint Analysis with Targeted Control-Flow Propagation ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. 2011 (cf. p. 84).
- [200] R. KANNAVARA, C. J. HAVLICEK, B. CHEN, M. R. TUTTLE, K. CONG, S. RAY et F. XIE. « Challenges and opportunities with concolic testing ». In : *2015 National Aerospace and Electronics Conference (NAECON)*. Juin 2015, p. 374-378 (cf. p. 46).
- [201] U. KARGEN et N. SHAHMEHRI. « Turning programs against each other : high coverage fuzz-testing using binary-code mutation and dynamic slicing ». In : *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. [PDF]. 2015, p. 782-792 (cf. p. 142, 192, 240).
- [202] H. KARIO. *TLS Test Framework*. [slides]. 2017 (cf. p. 146).
- [203] V. KASHYAP, J. RUCHTI, L. KOT, E. TURETSKY, R. SWORDS, D. MELSKI et E. SCHULTE. « Automated Customized Bug-Benchmark Generation ». In : *CoRR* abs/1901.02819 (2019). arXiv : [1901.02819](https://arxiv.org/abs/1901.02819) (cf. p. 22).
- [204] V. P. KEMERLIS, G. PORTOKALIDIS, K. JEE et A. D. KEROMYTIS. « libdft : practical dynamic data flow tracking for commodity systems ». In : *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*. 2012, p. 121-132 (cf. p. 211, 258).
- [205] T. V. KHANH, X. VU et M. OGAWA. « raSAT : SMT for Polynomial Inequality ». In : *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014, affiliated with the 26th International Conference on Computer Aided Verification (CAV 2014), the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), and the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014), Vienna, Austria, July 17-18, 2014*. 2014, p. 67 (cf. p. 69).
- [206] S. KIM, M. FAEREVAAG, M. JUNG, S. JUNG, D. OH, J. LEE et S. K. CHA. « Testing Intermediate Representations for Binary Analysis ». In : *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017, p. 353-364 (cf. p. 52).

## BIBLIOGRAPHIE

---

- [207] S. Y. KIM, S. LEE, I. YUN, W. XU, B. LEE, Y. YUN et T. KIM. « CAB-Fuzz : Practical Concolic Testing Techniques for COTS Operating Systems ». In : *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. [\[PDF\]](#) [\[slides\]](#) [\[audio\]](#). 2017, p. 689-701 (cf. p. 192).
- [208] J. C. KING. « Symbolic Execution and Program Testing ». In : *Communications of the ACM* 19.7 (juill. 1976), p. 385-394 (cf. p. 45, 57, 261).
- [209] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI. « Frama-C : A software analysis perspective ». In : *Formal Asp. Comput.* 27.3 (2015), p. 573-609 (cf. p. 48, 216, 243).
- [210] A. KISS, J. JASZ, G. LEHOTAI et T. GYIMOTHY. « Interprocedural static slicing of binary executables ». In : *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. [\[PDF\]](#). Sept. 2003, p. 118-127 (cf. p. 236).
- [211] G. KLEES, A. RUEF, B. COOPER, S. WEI et M. HICKS. « Evaluating Fuzz Testing ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. [\[PDF\]](#). 2018, p. 2123-2138 (cf. p. 17, 24).
- [212] A. KONOVALOV. *Coverage-Guided USB Fuzzing with Syzkaller*. [\[slides\]](#) [\[code\]](#). 15 fév. 2019 (cf. p. 136).
- [213] B. KOREL. « Automated Software Test Data Generation ». In : *IEEE Trans. Software Eng.* 16.8 (1990), p. 870-879 (cf. p. 218).
- [214] B. KOREL et J. W. LASKI. « Dynamic Program Slicing ». In : *Inf. Process. Lett.* 29.3 (1988), p. 155-163 (cf. p. 238).
- [215] J. KORET. *Nightmare : A distributed fuzzing testing suite with web administration*. [\[code\]](#). 2014 (cf. p. 149).
- [216] K. KU, T. E. HART, M. CHECHIK et D. LIE. « A buffer overflow benchmark for software model checkers ». In : *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 2007, p. 389-392 (cf. p. 18).
- [217] V. KUZNETSOV, J. KINDER, S. BUCUR et G. CANDEA. « Efficient state merging in symbolic execution ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 2012, p. 193-204 (cf. p. 65).
- [218] M. LABS. *KernelFuzzer* (cf. p. 135).
- [219] LAFINTEL. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. <https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/>. 2016 (cf. p. 139, 149, 220).
- [220] Z. LAI, S. CHEUNG et W. K. CHAN. « Detecting atomic-set serializability violations in multithreaded programs through active randomized testing ». In : *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. [\[PDF\]](#). 2010, p. 235-244 (cf. p. 149).

- [221] W. LANDI. « Undecidability of Static Analysis ». In : *LOPLAS 1.4* (1992), p. 323-337 (cf. p. 234).
- [222] M. A. LAURENZANO, M. M. TIKIR, L. CARRINGTON et A. SNAVELY. « PEBIL : Efficient static binary instrumentation for Linux ». In : *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2010, p. 175-183 (cf. p. 132, 133).
- [223] W. LE. « Segmented symbolic analysis ». In : *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, p. 212-221 (cf. p. 219).
- [224] *LeakSanitizer*. <https://clang.llvm.org/docs/LeakSanitizer.html> (cf. p. 140, 168).
- [225] J.-C. LECHENET. « Algorithmes certifiés pour la simplification syntaxique de programmes ». Theses. Université Paris-Saclay, juill. 2018 (cf. p. 231, 239).
- [226] B. LEE, C. SONG, T. KIM et W. LEE. « Type Casting Verification : Stopping an Emerging Attack Vector ». In : *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C. : USENIX Association, 2015, p. 81-96 (cf. p. 140).
- [227] C. LEE, M. POTKONJAK et W. H. MANGIONE-SMITH. « MediaBench : a tool for evaluating and synthesizing multimedia and communications systems ». In : *Proceedings of 30th Annual International Symposium on Microarchitecture*. Déc. 1997, p. 330-335 (cf. p. 18).
- [228] S. LEE, C. YOON, C. LEE, S. SHIN, V. YEGNESWARAN et P. A. PORRAS. « DELTA : A Security Assessment Framework for Software-Defined Networks ». In : *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. [PDF] [slide]. 2017 (cf. p. 149).
- [229] W. K. LEE, I. S. CHUNG, G. S. YOON et Y. R. KWON. « Specification-based program slicing and its applications ». In : *Journal of Systems Architecture* 47.5 (2001), p. 427-443 (cf. p. 239).
- [230] C. LEMIEUX, R. PADHYE, K. SEN et D. SONG. « PerfFuzz : automatically generating pathological inputs ». In : *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. [PDF] [code]. 2018, p. 254-265 (cf. p. 149).
- [231] C. LEMIEUX et K. SEN. « FairFuzz : a targeted mutation strategy for increasing greybox fuzz testing coverage ». In : *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. [PDF] [code]. 2018, p. 475-485 (cf. p. 149).
- [232] V. LEVENSHTEIN. « Binary Codes Capable of Correcting Deletions, Insertions and Reversals ». In : *Soviet Physics Doklady* 10 (fév. 1966). [PDF], p. 707 (cf. p. 176).
- [233] K. LI. *AFL's : Blindspot and How to Resist AFL Fuzzing for Arbitrary ELF Binaries*. [slides] [video]. 4 août 2018 (cf. p. 162).

## BIBLIOGRAPHIE

---

- [234] Y. LI, Z. SU, L. WANG et X. LI. « Steering symbolic execution to less traveled paths ». In : *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 2013, p. 19-32 (cf. p. 62).
- [235] Y. LI, B. CHEN, M. CHANDRAMOHAN, S. LIN, Y. LIU et A. TIU. « Steelix : program-state based binary fuzzing ». In : *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 2017, p. 627-637 (cf. p. 139, 149, 220).
- [236] Y. LI, S. JI, C. LV, Y. CHEN, J. CHEN, Q. GU et C. WU. *V-Fuzz : Vulnerability-Oriented Evolutionary Fuzzing*. [PDF]. 2019 (cf. p. 136).
- [237] C. LIAO, D. J. QUINLAN, R. W. VUDUC et T. PANAS. « Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization ». In : *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*. 2009, p. 308-322 (cf. p. 50).
- [238] C. LIDBURY, A. LASCU, N. CHONG et A. F. DONALDSON. « Many-core compiler fuzzing ». In : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. [PDF]. 2015, p. 65-76 (cf. p. 149).
- [239] Z. LIN et X. ZHANG. « Deriving Input Syntactic Structure from Execution ». In : *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. Atlanta, Georgia : ACM, 2008, p. 83-93 (cf. p. 136).
- [240] A. LTD. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. [manuel]. ARM Ltd. Avr. 2019 (cf. p. 310).
- [241] C.-K. LUK et al. « Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation ». In : *SIGPLAN Not.* 40.6 (juin 2005), p. 190-200 (cf. p. 105, 111, 132, 258).
- [242] LUNGTECH. *CGC : Cyber Grand Challenge (corpus)*. <http://www.lungetech.com/cgc-corpus/>. 2016 (cf. p. 11).
- [243] K. MA, Y. P. KHOO, J. S. FOSTER et M. HICKS. « Directed Symbolic Execution ». In : *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. 2011, p. 95-111 (cf. p. 63, 64, 194, 259).
- [244] L. MA, C. ARTHO, C. ZHANG, H. SATO, J. GMEINER et R. RAMLER. « GRT : Program-Analysis-Guided Random Testing (T) ». In : *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. [PDF]. 2015, p. 212-223 (cf. p. 191, 192).
- [245] R. MAJUMDAR et K. SEN. « Hybrid Concolic Testing ». In : *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. 2007, p. 416-426 (cf. p. 189).

- [246] V. J. M. MANÈS, H. HAN, C. HAN, S. K. CHA, M. EGELE, E. J. SCHWARTZ et M. WOO. « Fuzzing : Art, Science, and Engineering ». In : *CoRR* abs/1812.00140 (2018). arXiv : [1812.00140](#) (cf. p. 130, 148, 190).
- [247] A. MANGEAN, J. BÉCHENNEC, M. BRIDAY et S. FAUCOU. « BEST : a Binary Executable Slicing Tool ». In : *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France.* 2016, 7 :1-7 :10 (cf. p. 239).
- [248] L. MARTIGNONI, S. MCCAMANT, P. POOSANKAM, D. SONG et P. MANIATIS. « Path-exploration lifting : hi-fi tests for lo-fi emulators ». In : *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012.* 2012, p. 337-348 (cf. p. 73).
- [249] I. MASTROENI et D. ZANARDINI. « Abstract Program Slicing : An Abstract Interpretation-Based Approach to Program Slicing ». In : *ACM Trans. Comput. Log.* 18.1 (2017), 7 :1-7 :58 (cf. p. 233).
- [250] B. MATHIS, R. GOPINATH, M. MERA, A. KAMPMANN, M. HÖSCHELE et A. ZELLER. « Parser-Directed Fuzzing ». In : *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '19. Phoenix, USA : ACM, 2019 (cf. p. 149).
- [251] B. P. MILLER, L. FREDRIKSEN et B. SO. « An Empirical Study of the Reliability of UNIX Utilities ». In : *Commun. ACM* 33.12 (1990), p. 32-44 (cf. p. 9, 129).
- [252] D. MOLNAR, X. C. LI et D. A. WAGNER. « Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs ». In : *Proceedings of the 18th Conference on USENIX Security Symposium.* SSYM'09. Montreal, Canada : USENIX Association, 2009, p. 67-82 (cf. p. 142).
- [253] A. MONEGER. *Scapy TLS : a scriptable TLS 1.3 stack.* [\[slides\]](#). 2017 (cf. p. 145, 146).
- [254] L. MOONEN. « Generating Robust Parsers Using Island Grammars ». In : *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001.* 2001, p. 13 (cf. p. 23).
- [255] C. MOUGEY et F. DESCLAUX. *Miasm : Reverse Engineering Framework.* [\[slide\]](#). 4 août 2018 (cf. p. 49, 50, 52, 73).
- [256] L. M. de MOURA et N. BJØRNER. « Satisfiability modulo theories : introduction and applications ». In : *Commun. ACM* 54.9 (2011), p. 69-77 (cf. p. 66).
- [257] L. M. de MOURA et N. BJØRNER. « Z3 : An Efficient SMT Solver ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 2008, p. 337-340 (cf. p. 68, 69, 89, 96, 105).
- [258] M. MUENCH, D. NISI, A. FRANCILLON et D. BALZAROTTI. « Avatar 2 : A Multi-target Orchestration Platform ». In : [\[PDF\]](#). 2018 (cf. p. 55, 105).

## BIBLIOGRAPHIE

---

- [259] G. B. MUND, R. MALL et S. SARKAR. « Computation of intraprocedural dynamic program slices ». In : *Information & Software Technology* 45.8 (2003), p. 499-512 (cf. p. 238).
- [260] S. NAGARAKATTE, J. ZHAO, M. M. MARTIN et S. ZDANCEWIC. « SoftBound : Highly Compatible and Complete Spatial Memory Safety for C ». In : *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland : ACM, 2009, p. 245-258 (cf. p. 140).
- [261] S. NAGARAKATTE, J. ZHAO, M. M. MARTIN et S. ZDANCEWIC. « CETS : Compiler Enforced Temporal Safety for C ». In : *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. Toronto, Ontario, Canada : ACM, 2010, p. 31-40 (cf. p. 140).
- [262] S. NAGY et M. HICKS. « Full-speed Fuzzing : Reducing Fuzzing Overhead through Coverage-guided Tracing ». In : *CoRR* abs/1812.11875 (2018). [\[PDF\]](#) [\[code\]](#) [\[video\]](#). arXiv : [1812.11875](#) (cf. p. 149, 259).
- [263] G. C. NECULA, S. MCPEAK, S. P. RAHUL et W. WEIMER. « CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs ». In : *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 2002, p. 213-228 (cf. p. 48, 50).
- [264] N. NETHERCOTE et J. SEWARD. « Valgrind : a framework for heavyweight dynamic binary instrumentation ». In : *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, p. 89-100 (cf. p. 48-50, 101, 132, 133).
- [265] S. NILIZADEH, Y. NOLLER et C. S. PASAREANU. « DiffFuzz : Differential Fuzzing for Side-Channel Analysis ». In : *Proceedings of 41th International Conference on Software Engineering*. [\[PDF\]](#) [\[code\]](#). 2019 (cf. p. 141).
- [266] R. J. NSA. *Get Your Free NSA Reverse Engineering Tool*. [\[PDF\]](#). 4 mars 2019 (cf. p. 49, 50).
- [267] S. OGNAWALA, T. HUTZELMANN, E. PSALLIDA et A. PRETSCHNER. « Improving Function Coverage with Munch : A Hybrid Fuzzing and Directed Symbolic Execution Approach ». In : *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France : ACM, 2018, p. 1475-1482. eprint : \tt [\[PDF\]](#) (cf. p. 194).
- [268] S. OGNAWALA, F. KILGER et A. PRETSCHNER. « Compositional Fuzzing Aided by Targeted Symbolic Execution ». In : *CoRR* abs/1903.02981 (2019). [\[PDF\]](#). arXiv : [1903.02981](#) (cf. p. 194).
- [269] S. OGNAWALA, M. OCHOA, A. PRETSCHNER et T. LIMMER. « MACKE : compositional analysis of low-level vulnerabilities with symbolic execution ». In : *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 2016, p. 780-785 (cf. p. 189, 194).

- [270] K. J. OTTENSTEIN et L. M. OTTENSTEIN. « The Program Dependence Graph in a Software Development Environment ». In : *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*. 1984, p. 177-184 (cf. p. 232).
- [271] R. PADHYE, C. LEMIEUX, K. SEN, M. PAPADAKIS et Y. L. TRAON. « Zest : Validity Fuzzing and Parametric Generators for Effective Random Testing ». In : *CoRR abs/1812.00078* (2018). [\[PDF\]](#). arXiv : [1812.00078](#) (cf. p. 135).
- [272] J. PAN, G. YAN et X. FAN. « Digtool : A Virtualization-Based Framework for Detecting Kernel Vulnerabilities ». In : *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. [\[PDF\]](#) [\[slide\]](#) [\[video\]](#). 2017, p. 149-165 (cf. p. 149).
- [273] C. PARK et K. SEN. « Randomized active atomicity violation detection in concurrent programs ». In : *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. [\[PDF\]](#). 2008, p. 135-145 (cf. p. 149).
- [274] C. S. PASAREANU, N. RUNGTA et W. VISSER. « Symbolic execution with mixed concrete-symbolic solving ». In : *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 2011, p. 34-44 (cf. p. 54).
- [275] M. PAYER. « The Fuzzing Hype-Train : How Random Testing Triggers Thousands of Crashes ». In : *IEEE Security Privacy* 17.1 (jan. 2019), p. 78-82 (cf. p. 129).
- [276] H. PENG, Y. SHOSHITAISHVILI et M. PAYER. « T-Fuzz : Fuzzing by Program Transformation ». In : *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. [\[PDF\]](#) [\[code\]](#). 2018, p. 697-710 (cf. p. 191, 192, 260).
- [277] J. PEREYDA. *Boofuzz*. <https://github.com/jtpereyda/boofuzz> (cf. p. 145, 149).
- [278] D. M. PERRY, A. MATTAVELLI, X. ZHANG et C. CADAR. « Accelerating array constraints in symbolic execution ». In : *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 2017, p. 68-78 (cf. p. 70).
- [279] T. PETSIOS, A. TANG, S. STOLFO, A. D. KEROMYTIS et S. JANA. « NEZHA : Efficient Domain-Independent Differential Testing ». In : *2017 IEEE Symposium on Security and Privacy (SP)*. Mai 2017, p. 615-632 (cf. p. 141).
- [280] T. PETSIOS, J. ZHAO, A. D. KEROMYTIS et S. JANA. « SlowFuzz : Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017, p. 2155-2168 (cf. p. 149).

## BIBLIOGRAPHIE

---

- [281] J. PEWNY et T. HOLZ. « EvilCoder : automated bug insertion ». In : *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. 2016, p. 214-225 (cf. p. 22).
- [282] V. PHAM, M. BÖHME et A. ROYCHOUDHURY. « Model-based whitebox fuzzing for program binaries ». In : *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. [PDF]. 2016, p. 543-553 (cf. p. 192).
- [283] V.-T. PHAM, M. BÖHME, A. E. SANTOSA, A. R. CĂCIULESCU et A. ROYCHOUDHURY. *Smart Greybox Fuzzing*. [PDF] [code]. 2018. arXiv : 1811.09447 [cs.CR] (cf. p. 149).
- [284] X. QU et B. ROBINSON. « A Case Study of Concolic Testing Tools and their Limitations ». In : *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*. 2011, p. 117-126 (cf. p. 9, 46).
- [285] J. R. LYLE et D. BINKLEY. *Program Slicing in the Presence of Pointers*. Rapp. tech. [PDF], p. 12 (cf. p. 239).
- [286] C. R2. *Radare2*. <https://rada.re/r/index.html> (cf. p. 50, 52).
- [287] M. RAJPAL, W. BLUM et R. SINGH. « Not all bytes are equal : Neural byte sieve for fuzzing ». In : *CoRR* abs/1711.04596 (2017). arXiv : 1711.04596 (cf. p. 144).
- [288] D. A. RAMOS et D. R. ENGLER. « Under-Constrained Symbolic Execution : Correctness Checking for Real Code ». In : *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015, p. 49-64 (cf. p. 195, 209).
- [289] D. A. RAMOS et D. R. ENGLER. « Under-Constrained Symbolic Execution : Correctness Checking for Real Code ». In : *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. 2016 (cf. p. 70).
- [290] S. RAWAT, V. JAIN, A. KUMAR, L. COJOCAR, C. GIUFFRIDA et H. BOS. « VUzzer : Application-aware Evolutionary Fuzzing ». In : *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. [PDF] [code] [video]. 2017 (cf. p. 130, 143, 191, 192, 211, 220).
- [291] A. REBERT, S. K. CHA, T. AVGERINOS, J. FOOTE, D. WARREN, G. GRIECO et D. BRUMLEY. « Optimizing Seed Selection for Fuzzing ». In : *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, 2014, p. 861-875 (cf. p. 143, 220).
- [292] E. REISNER, C. SONG, K. MA, J. S. FOSTER et A. A. PORTER. « Using symbolic evaluation to understand behavior in configurable software systems ». In : *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. [PDF] [code]. 2010, p. 445-454 (cf. p. 48, 73).

- [293] T. W. REPS. « Program analysis via graph reachability ». In : *Information & Software Technology* 40.11-12 (1998), p. 701-726 (cf. p. 233).
- [294] T. W. REPS, S. HORWITZ, S. SAGIV et G. ROSAY. « Speeding up Slicing ». In : *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 6-9, 1994*. 1994, p. 11-20 (cf. p. 236).
- [295] T. W. REPS et G. ROSAY. « Precise Interprocedural Chopping ». In : *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*. 1995, p. 41-52 (cf. p. 232).
- [296] H. RIENER, M. SOEKEN, C. WERTHER, G. FEY et R. DRECHSLER. « MetaSMT : a unified interface to SMT-LIB2 ». In : *Proceedings of the 2014 Forum on Specification and Design Languages, FDL 2014, Munich, Germany, October 14-16, 2014*. 2014, p. 1-6 (cf. p. 68, 96).
- [297] S. ROY, A. PANDEY, B. DOLAN-GAVITT et Y. HU. « Bug synthesis : challenging bug-finding tools with deep faults ». In : *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 2018, p. 224-234 (cf. p. 22).
- [298] J. RUDERMAN. *Lithium* (cf. p. 142).
- [299] J. de RUITER et E. POLL. « Protocol State Fuzzing of TLS Implementations ». In : *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015, p. 193-206 (cf. p. 146).
- [300] D. RUTISHAUSER. *Fuzzing For Worms*. [\[slides\]](#) (cf. p. 147-149, 164).
- [301] A. Z. SALAMON. « Transformations of representation in constraint satisfaction ». Thèse de doct. 2013 (cf. p. 311).
- [302] J. SALWAN, S. BARDIN et M. POTET. « Symbolic Deobfuscation : From Virtualized Code Back to the Original ». In : *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*. 2018, p. 372-392 (cf. p. 56).
- [303] M. SAMAK, M. K. RAMANATHAN et S. JAGANNATHAN. « Synthesizing racy tests ». In : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. [\[PDF\]](#). 2015, p. 175-185 (cf. p. 192).
- [304] *Triton : A Dynamic Symbolic Execution Framework*. SSTIC, 2015, p. 31-54 (cf. p. 8, 58, 68, 73, 77, 142, 243, 256).
- [305] P. SAXENA, S. HANNA, P. POOSANKAM et D. SONG. « FLAX : Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications ». In : *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. [\[PDF\]](#). 2010 (cf. p. 149).

## BIBLIOGRAPHIE

---

- [306] S. SCHUMILO, C. ASCHERMANN, R. GAWLIK, S. SCHINZEL et T. HOLZ. « kAFL : Hardware-Assisted Feedback Fuzzing for OS Kernels ». In : *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* 2017, p. 167-182 (cf. p. 149).
- [307] E. J. SCHWARTZ, T. AVGERINOS et D. BRUMLEY. « All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) ». In : *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA.* 2010, p. 317-331 (cf. p. 56).
- [308] M. SECURITY. *jsfunfuzz* (cf. p. 135, 149).
- [309] K. SEN. « Effective random testing of concurrent programs ». In : *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA.* 2007, p. 323-332 (cf. p. 149).
- [310] K. SEN. « Race directed random testing of concurrent programs ». In : *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* [PDF]. 2008, p. 11-21 (cf. p. 149).
- [311] K. SEN, D. MARINOV et G. AGHA. « CUTE : a concolic unit testing engine for C ». In : *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005.* [PDF]. 2005, p. 263-272 (cf. p. 45, 58, 60, 73).
- [312] K. SEREBRYANY, D. BRUENING, A. POTAPENKO et D. VYUKOV. « AddressSanitizer : A Fast Address Sanity Checker ». In : *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12).* [PDF] [code]. Boston, MA : USENIX, 2012, p. 309-318 (cf. p. 140, 159, 168).
- [313] K. SEREBRYANY et T. ISKHODZHANOV. « ThreadSanitizer : Data Race Detection in Practice ». In : *Proceedings of the Workshop on Binary Instrumentation and Applications.* WBIA '09. New York, New York, USA : ACM, 2009, p. 62-71 (cf. p. 141, 168).
- [314] M. SHAHBAZ et R. GROZ. « Inferring Mealy Machines ». In : *FM 2009 : Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings.* 2009, p. 207-222 (cf. p. 146).
- [315] A. SHARMA. « Exploiting Undefined Behaviors for Efficient Symbolic Execution ». In : *Companion Proceedings of the 36th International Conference on Software Engineering.* ICSE Companion 2014. [PDF] [code]. Hyderabad, India : ACM, 2014, p. 727-729 (cf. p. 73).
- [316] D. SHE, K. PEI, D. EPSTEIN, J. YANG, B. RAY et S. JANA. « NEUZZ : Efficient Fuzzing with Neural Program Learning ». In : *CoRR abs/1807.05620* (2018). [PDF] [code]. arXiv : 1807.05620 (cf. p. 136).

- [317] S. SHIRAISHI, V. MOHAN et H. MARIMUTHU. « Test suites for benchmarks of static analysis tools ». In : *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*. 2015, p. 12-15 (cf. p. 18).
- [318] D. R. SHIREY. *RFC 2828*. Mai 2000 (cf. p. 12).
- [319] Y. SHOSHITAISHVILI, R. WANG, C. HAUSER, C. KRUEGEL et G. VIGNA. « Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware ». In : *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. 2015 (cf. p. 195).
- [320] Y. SHOSHITAISHVILI et al. « SOK : (State of) The Art of War : Offensive Techniques in Binary Analysis ». In : *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 2016, p. 138-157 (cf. p. 11, 45, 52, 53, 55, 68, 73, 115, 199, 208, 243).
- [321] Y. SHOSHITAISHVILI et al. « Mechanical Phish : Resilient Autonomous Hacking ». In : *IEEE Security & Privacy* 16.2 (2018), p. 12-22 (cf. p. 100, 199).
- [322] S. SINHA, M. J. HARROLD et G. ROTHERMEL. « System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow ». In : *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*. 1999, p. 432-441 (cf. p. 238).
- [323] J. SLABÝ, J. STREJČEK et M. TRTÍK. « Checking Properties Described by State Machines : On Synergy of Instrumentation, Slicing, and Symbolic Execution ». In : *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*. 2012, p. 207-221 (cf. p. 8, 194, 239, 240, 243, 248).
- [324] « SMT Solvers in Software Security ». In : *Presented as Part of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA : USENIX, 2012 (cf. p. 66).
- [325] J. SOMOROVSKY. « Systematic Fuzzing and Testing of TLS Libraries ». In : *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, p. 1492-1504 (cf. p. 146, 149).
- [326] D. X. SONG et al. « BitBlaze : A New Approach to Computer Security via Binary Analysis ». In : *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*. [PDF] [page]. 2008, p. 1-25 (cf. p. 45, 49, 50, 73).
- [327] D. SONG et al. « PeriScope : An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary ». In : *Network and Distributed System Security Symposium (NDSS)*. [PDF] [slide] [video] [code]. 2019 (cf. p. 149).
- [328] J. SONG et J. ALVES-FOSS. « The DARPA Cyber Grand Challenge : A Competitor's Perspective, Part 2 ». In : *IEEE Security Privacy* 14.1 (jan. 2016), p. 76-81 (cf. p. 193).

## BIBLIOGRAPHIE

---

- [329] S. SPARKS, S. EMBLETON, R. CUNNINGHAM et C. C. ZOU. « Automated Vulnerability Analysis : Leveraging Control Flow for Evolutionary Input Crafting ». In : *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. [PDF]. 2007, p. 477-486 (cf. p. 149).
- [330] A. SRIVASTAVA, A. EDWARDS et H. VO. *Vulcan : Binary Transformation In A Distributed Environment*. Rapp. tech. 2001, p. 12 (cf. p. 132, 133).
- [331] C. STATION. *Cranelift Code Generator*. <https://cranelift.readthedocs.io/en/latest/index.html>. 2019 (cf. p. 49, 50).
- [332] E. STEPANOV et K. SEREBRYANY. « MemorySanitizer : Fast detector of uninitialized memory use in C++ ». In : *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [code]. Fév. 2015, p. 46-55 (cf. p. 141, 159).
- [333] N. STEPHENS et al. « Driller : Augmenting Fuzzing Through Selective Symbolic Execution ». In : *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. 2016 (cf. p. 8, 101, 190, 192, 195, 197, 208, 211, 256).
- [334] Y. SUI et J. XUE. « SVF : interprocedural static value-flow analysis in LLVM ». In : *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. [PDF]. 2016, p. 265-266 (cf. p. 240).
- [335] R. SWIECKI et F. GRÖBERT. *honggfuzz*. <https://github.com/google/honggfuzz> (cf. p. 8, 17, 129, 130, 137, 143, 149, 256).
- [336] SYNOPSYS. *Defensics Fuzz Testing (ex :Codenomicon*. <https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html> (cf. p. 149).
- [337] L. SZEKERES. « Memory corruption mitigation via hardening and testing ». [thesis]. Thèse de doct. Stony Brook University, mai 2017 (cf. p. 217).
- [338] L. TEAM. *libFuzzer – a library for coverage-guided fuzz testing*. <https://llvm.org/docs/LibFuzzer.html> (cf. p. 17, 129, 137, 149).
- [339] L. development TEAM. *LLVM Language Reference Manual*. <https://llvm.org/docs/LangRef.html> (cf. p. 50).
- [340] TETRANE. *Reven - Tetrane*. <https://www.tetrane.com> (cf. p. 60, 73).
- [341] A. THAKUR et al. « Directed Proof Generation for Machine Code ». In : *Proceedings of the 22Nd International Conference on Computer Aided Verification*. CAV'10. [PDF]. Edinburgh, UK : Springer-Verlag, 2010, p. 288-305 (cf. p. 73).
- [342] C. S. TIMPERLEY, S. STEPNEY et C. LE GOUES. « BugZoo : a platform for studying software bugs ». In : *Proceedings of the 40th International Conference on Software Engineering : Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 2018, p. 446-447 (cf. p. 21).
- [343] F. TIP. *A Survey of Program Slicing Techniques*. Rapp. tech. Amsterdam, The Netherlands, The Netherlands : CWI (Centre for Mathematics et Computer Science), 1994 (cf. p. 231).

- [344] L. D. TOFFOLA, C. STAICU et M. PRADEL. « Saying ‘Hi!’ is not enough : Mining inputs for effective test generation ». In : *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, p. 44-49 (cf. p. 136).
- [345] D. TRABISH, A. MATTAVELLI, N. RINETZKY et C. CADAR. « Chopped symbolic execution ». In : *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. [PDF]. 2018, p. 350-360 (cf. p. 192, 240, 243, 252, 257).
- [346] N. J. TRACEY, J. A. CLARK, K. MANDER et J. A. McDERMID. « An Automated Framework for Structural Test-Data Generation ». In : *The Thirteenth IEEE Conference on Automated Software Engineering, ASE 1998, Honolulu, Hawaii, USA, October 13-16, 1998*. 1998, p. 285-288 (cf. p. 218).
- [347] M. TRTÍK et J. STREJCEK. « Symbolic Memory with Pointers ». In : *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. 2014, p. 380-395 (cf. p. 56, 57).
- [348] C. U.S. *Basic Fuzzing Framework*. <https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework>. 2016 (cf. p. 130, 137, 142, 149).
- [349] C. U.S. *Failure Observation Engine*. [code] (cf. p. 137, 149).
- [350] G. A. VENKATESH. « The Semantic Approach to Program Slicing ». In : *Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. 1991, p. 107-119 (cf. p. 231, 239).
- [351] W. VISSER, J. GELDENHUYSEN et M. B. DWYER. « Green : reducing, reusing and recycling constraints in program analysis ». In : *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE’12, Cary, NC, USA - November 11 - 16, 2012*. 2012, p. 58 (cf. p. 69).
- [352] D. VYUKOV. *syzkaller* (cf. p. 136, 149).
- [353] J. WANG, B. CHEN, L. WEI et Y. LIU. « Skyfire : Data-Driven Seed Generation for Fuzzing ». In : *2017 IEEE Symposium on Security and Privacy (SP)*. [PDF] [code]. Mai 2017, p. 579-594 (cf. p. 136, 148).
- [354] J. WANG, B. CHEN, L. WEI et Y. LIU. « Superion : Grammar-Aware Greybox Fuzzing ». In : *Proceedings of 41th International Conference on Software Engineering*. [PDF] [code]. 2019 (cf. p. 149).
- [355] S. WANG, J. NAM et L. TAN. « QTEP : Quality-aware Test Case Prioritization ». In : *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany : ACM, 2017, p. 523-534 (cf. p. 134).

## BIBLIOGRAPHIE

---

- [356] T. WANG, T. WEI, G. GU et W. ZOU. « TaintScope : A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection ». In : *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*. [PDF] [slides]. 2010, p. 497-512 (cf. p. 191, 192).
- [357] X. WANG, J. SUN, Z. CHEN, P. ZHANG, J. WANG et Y. LIN. « Towards optimal concolic testing ». In : *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. [PDF]. 2018, p. 291-302 (cf. p. 62).
- [358] M. P. WARD et H. ZEDAN. « Slicing as a program transformation ». In : *ACM Trans. Program. Lang. Syst.* 29.2 (2007), p. 7 (cf. p. 231).
- [359] D. WASSERRAB, D. LOHNER et G. SNELTING. « On PDG-based noninterference and its modular proof ». In : *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. 2009, p. 31-44 (cf. p. 231).
- [360] D. WATSON. *libunwind, portable and efficient C programming interface (API) to determine the call-chain of a program*. [code] (cf. p. 168).
- [361] M. WEISER. « Program Slicing ». In : *IEEE Trans. Software Eng.* 10.4 (1984), p. 352-357 (cf. p. 231, 232).
- [362] M. WEISER et J. LYLE. « Experiments on Slicing-based Debugging Aids ». In : *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Washington, D.C., USA : Ablex Publishing Corp., 1986, p. 187-197 (cf. p. 232).
- [363] S.-H. WEN, W.-L. MOW, W.-N. CHEN, C.-Y. WANG et H.-C. HSIAO. « Enhancing Symbolic Execution by Machine Learning Based Solver Selection ». In : *Proceedings of the NDSS Workshop on Binary Analysis Research*. [PDF] [code]. 2019 (cf. p. 67).
- [364] N. WILLIAMS, B. MARRE, P. MOUY et M. ROGER. « PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis ». In : *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*. 2005, p. 281-292 (cf. p. 45, 48, 73).
- [365] M. WOO, S. K. CHA, S. GOTTLIEB et D. BRUMLEY. « Scheduling Black-box Mutational Fuzzing ». In : *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany : ACM, 2013, p. 511-522 (cf. p. 134, 142, 145, 149).
- [366] T. XIE, N. TILLMANN, J. de HALLEUX et W. SCHULTE. « Fitness-guided path exploration in dynamic symbolic execution ». In : *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. 2009, p. 359-368 (cf. p. 63).

- [367] X. XIE, B. CHEN, Y. LIU, W. LE et X. LI. « Proteus : computing disjunctive loop summary via path dependency analysis ». In : *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016.* 2016, p. 61-72 (cf. p. 65).
- [368] Y. XIE, A. CHOU et D. R. ENGLER. « ARCHER : using symbolic, path-sensitive analysis to detect memory access errors ». In : *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003.* 2003, p. 327-336 (cf. p. 219).
- [369] B. XU, J. QIAN, X. ZHANG, Z. WU et L. CHEN. « A brief survey of program slicing ». In : *ACM SIGSOFT Software Engineering Notes* 30.2 (2005), p. 1-36 (cf. p. 231).
- [370] H. XU, Z. ZHAO, Y. ZHOU et M. R. LYU. « Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs ». In : *IEEE Transactions on Dependable and Secure Computing* (2018), p. 1-1 (cf. p. 19, 24, 27).
- [371] W. XU, S. KASHYAP, C. MIN et T. KIM. « Designing New Operating Primitives to Improve Fuzzing Performance ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA : ACM, 2017, p. 2313-2328 (cf. p. 138).
- [372] F. YAMAGUCHI, N. GOLDE, D. ARP et K. RIECK. « Modeling and Discovering Vulnerabilities with Code Property Graphs ». In : *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* 2014, p. 590-604 (cf. p. 23).
- [373] F. YAMAGUCHI, N. GOLDE, D. ARP et K. RIECK. « Modeling and Discovering Vulnerabilities with Code Property Graphs ». In : *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* 2014, p. 590-604 (cf. p. 261).
- [374] D. YANG, Y. ZHANG et Q. LIU. « BlendFuzz : A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs ». In : *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. 2012, p. 1070-1076 (cf. p. 135).
- [375] G. YANG, C. S. PASAREANU et S. KHURSHID. « Memoized symbolic execution ». In : *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012.* 2012, p. 144-154 (cf. p. 69).
- [376] Q. YI, Z. YANG, S. GUO, C. WANG, J. LIU et C. ZHAO. « Postconditioned Symbolic Execution ». In : *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015.* 2015, p. 1-10 (cf. p. 11).
- [377] W. YOU, X. LIU, S. MA, D. PERRY, X. ZHANG et B. LIANG. « SLF : Fuzzing without Valid Seed Inputs ». In : *Proceedings of 41th International Conference on Software Engineering*. [PDF]. 2019 (cf. p. 149, 195).

## BIBLIOGRAPHIE

---

- [378] W. YOU, X. WANG, S. MA, J. HUANG, X. ZHANG, X. WANG et B. LIANG. « Pro-Fuzzer : On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery ». In : *IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 20-22, 2019*. [PDF]. 2019 (cf. p. 136, 149).
- [379] H. YU, Z. CHEN, J. WANG, Z. SU et W. DONG. « Symbolic verification of regular properties ». In : *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 2018, p. 871-881 (cf. p. 63).
- [380] I. YUN, S. LEE, M. XU, Y. JANG et T. KIM. « QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing ». In : *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD : USENIX Association, 2018, p. 745-761 (cf. p. 8, 49, 122, 192, 195, 208, 217, 258, 308).
- [381] M. ZALEWSKI. *AFL Dictionnaires*. <https://github.com/rc0r/afl-fuzz/tree/master/dictionaries> (cf. p. 158).
- [382] M. ZALEWSKI. *AFL Notes for ASan*. [https://github.com/mirrorer/afl/blob/master/docs/notes\\_for\\_asan.txt](https://github.com/mirrorer/afl/blob/master/docs/notes_for_asan.txt) (cf. p. 159).
- [383] M. ZALEWSKI. *American Fuzzy Lop*. <http://lcamtuf.coredump.cx/afl/> (cf. p. 17, 129, 133, 137, 149, 156, 201).
- [384] M. ZALEWSKI. *Crossfuzz* (cf. p. 135).
- [385] M. ZALEWSKI. *Fuzzing random programs without execve()*. <https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html> (cf. p. 157).
- [386] M. ZALEWSKI. *New in AFL : persistent mode*. <https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html> (cf. p. 158).
- [387] M. ZALEWSKI. *Technical "whitepaper" for afl-fuzz*. [http://lcamtuf.coredump.cx/afl/technical\\_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt) (cf. p. 156, 157).
- [388] M. ZALEWSKI. *Tips for parallel fuzzing*. [https://github.com/mirrorer/afl/blob/master/docs/parallel\\_fuzzing.txt](https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt) (cf. p. 157).
- [389] A. ZELLER et R. HILDEBRANDT. « Simplifying and Isolating Failure-Inducing Input ». In : *IEEE Trans. Softw. Eng.* 28.2 (fév. 2002), p. 183-200 (cf. p. 142).
- [390] G. P. ZERO. *WinAFL* (cf. p. 138).
- [391] H. ZHANG, A. ZHOU, P. JIA, L. LIU, J. MA et L. LIU. « InsFuzz : Fuzzing Binaries With Location Sensitivity ». In : *IEEE Access* 7 (2019). [PDF], p. 22434-22444 (cf. p. 149).
- [392] M. ZHANG, R. QIAO, N. HASABNIS et R. SEKAR. « A Platform for Secure Static Binary Instrumentation ». In : *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '14. Salt Lake City, Utah, USA : ACM, 2014, p. 129-140 (cf. p. 132, 133).

- [393] Y. ZHANG, Z. CHEN, J. WANG, W. DONG et Z. LIU. « Regular Property Guided Dynamic Symbolic Execution ». In : *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015, p. 643-653 (cf. p. 63).
- [394] L. ZHAO, Y. DUAN, H. YIN et J. XUAN. « Send Hardest Problems My Way : Probabilistic Path Prioritization for Hybrid Fuzzing ». In : *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. [PDF] [slide] [video]. 2019 (cf. p. 190, 192).
- [395] Y. ZHENG, X. ZHANG et V. GANESH. « Z3-str : a z3-based string solver for web application analysis ». In : *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 2013, p. 114-124 (cf. p. 68, 105).
- [396] N. N. ZISIS SIALVERAS. *Introducing Choronzon : An approach at knowledge-based evolutionary fuzzing*. [slides]. 2015 (cf. p. 132, 149).

*BIBLIOGRAPHIE*

---

## Annexe A

### Machines de Mealy

Les machines de Mealy sont couramment utilisées pour modéliser des machines à états protocolaires (*dans un contexte réseau*) d'un point de vue mathématique. Une machine de Mealy est un automate fini avec sorties et plus précisément un transducteur fini. Cette modélisation est plus primitive que les machines de Turing ou que les automates à piles dans la mesure où elle n'implique aucune modélisation de mémoire si c'est n'est les états eux-mêmes. La figure A.1 donne la hiérarchie des automates.

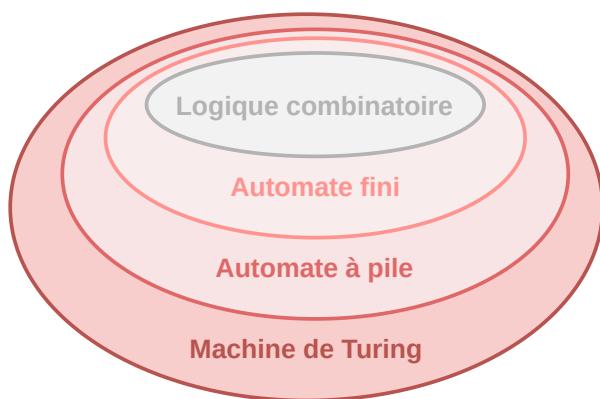


FIGURE A.1 – Hiérarchie des automates

Parmis les automates finis se distingue, les accepteurs et les transducteurs. Les accepteurs acceptent ou rejettent les données en entrée au regard des transitions possibles sur l'automate. Leur sortie est donc binaire, alors que les transducteurs génèrent un ensemble de mots sur un alphabet de sortie à partir de celui d'entrée.

Un autre exemple de transducteur fini sont les machines de Moore qui fournissent les mêmes propriétés que les machines de Mealy. Ainsi, il est possible de convertir n'importe quelle machine de Mealy en machine de Moore et vice versa. La différence se situe dans le fait que les messages de sorties sont associés aux états et non pas aux transitions (arêtes) de l'automate. Cela induit généralement plus d'états pour un automate de Moore.

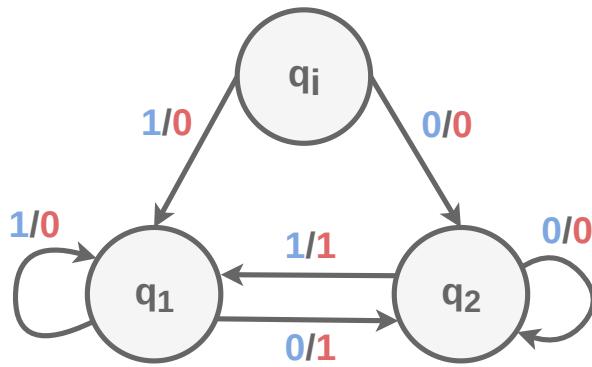


FIGURE A.2 – Machine de Mealy pour la fonction XOR

La figure A.2 représente une machine de mealy encodant la fonction XOR. À l'inverse d'une machine de Moore se sont les transitions qui portent le message d'entrée (en bleu) et le message de sortie émis (en rouge). Ainsi, en envoyant le message  $S_a \triangleq 00011011$  qui correspond à l'ensemble des valeurs possible de la table de vérité (00, 01, 10 et 11). On obtient le message de sortie 00010111 où le deuxième message pour chaque couple (-0-1-1-0) renvoie la valeur de retour du XOR.

## Annexe B

### Compétition Rode0day

Rode0day<sup>177</sup> est une compétition de recherche de vulnérabilité créée en collaboration entre le *Lincoln Laboratory* du *Massachusetts Institute of Technology (MIT)* et l'université de New York. Elle s'inspire fortement du CGC mais ouvre la participation à tout le monde. Elle est active en continu mais des "manches" sont jouées mensuellement avec la publication de nouveaux binaires.

L'ajout de vulnérabilités dans les binaires est effectué avec LAVA [109]. Le principe est donc similaire à la suite LAVA-M. Les binaires utilisés sont de vrais utilitaires comme `pcre`, `grep` ou `sqlite`. Les programmes fournis peuvent être de n'importe quelle architecture (mais généralement x86 ou ARM) et sous la forme de code source ou de binaires.

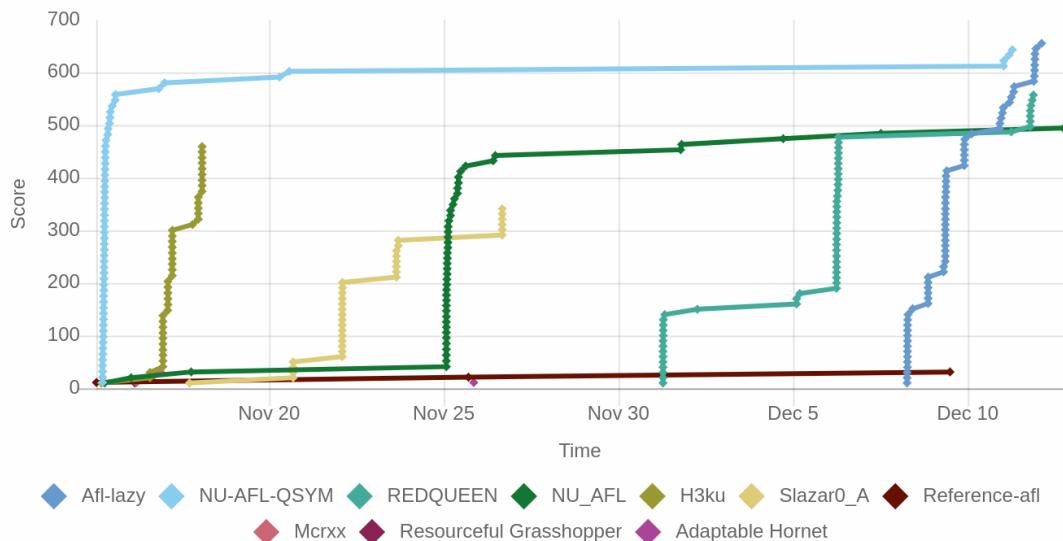


FIGURE B.1 – Résultats Rode0day- Novembre 2018

Concrètement, après inscription un identifiant d'utilisateur est fourni. Il est alors possible de récupérer les programmes via une API REST. Pour valider la découverte d'un

177. <https://rode0day.mit.edu>

## Annexe B. Compétition Rode0day

---

bug, il faut soumettre une entrée permettant de l'activer. La découverte d'un bug est validée à cette unique condition. Chaque bug unique trouvé rapporte 10 points plus 1 point si c'est le premier à le trouver parmi les compétiteurs.

Le seul biais que peut avoir la compétition est qu'aucune contrainte n'est imposée en termes de puissance de calcul (mais ce ne serait pas possible à mettre en oeuvre). L'information de puissance est laissée au bon vouloir des équipes qui le fournissent sur la page de l'équipe. Par exemple **NU-AFL-QSYM**<sup>178</sup> fonctionne avec 40 coeurs et 256 Go de RAM.

Cette compétition est actuellement la meilleure méthode pour évaluer de manière comparative les performances des outils existants. En effet, la plupart des outils de l'état de concurrence ou ont concouru par le passé. Les Figures B.1 et B.2 montrent respectivement les résultats pour le mois de novembre 2018 et de février 2019. Les équipes préfixé par "NU-" précise que l'outil n'a pas été modifié par rapport à la version originale. En plus de fournir des résultats quantitatifs les graphes permettent d'évaluer la rapidité avec laquelle les outils trouvent les bugs. Pour, un même nombre de bugs trouvés la rapidité, elle, diffère énormément. Par exemple, en février **afl-lazy** trouve presque tous les bugs en 3 jours alors qu'il faut 1 mois à **AFL** pour arriver au même résultat.

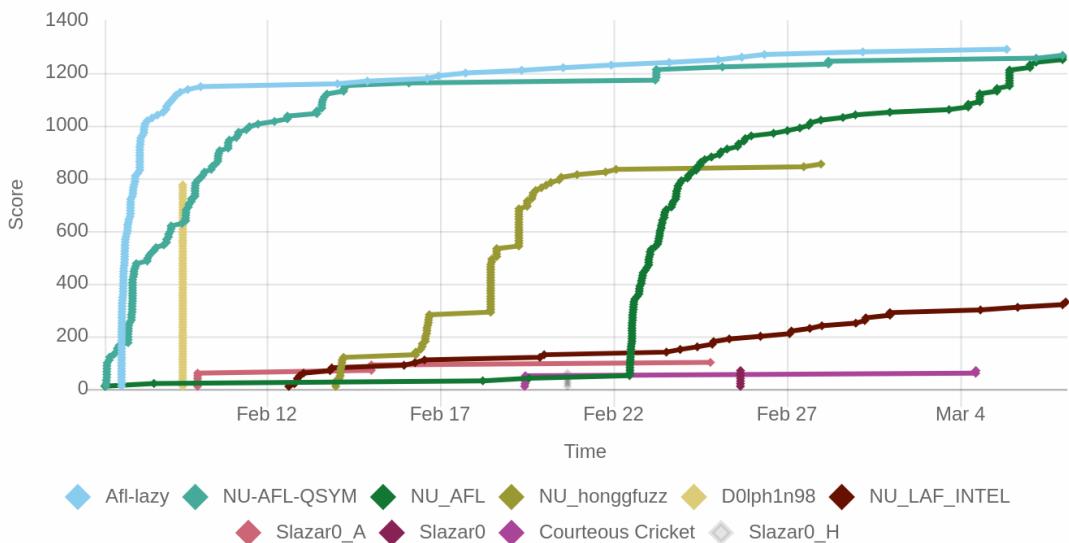


FIGURE B.2 – Résultats Rode0day- Février 2019

En sus, cette compétition permet d'estimer la qualité des outils, y compris ceux n'ayant pas encore été publiés comme **afl-lazy** ou qui ne sont pas *open-source* comme **redqueen** [13]. Ces deux outils fournissant avec **Qsym** [380] des résultats fracassants.

178. <https://rode0day.mit.edu/profile/NU-AFL-QSYM>

## Annexe C

### Sail : ISA ARMv8-A

Le projet **Sail**<sup>179</sup> [10] aspire à fournir une modélisation rigoureuse de la sémantique des instructions de différentes architectures (ISA) dans une même représentation intermédiaire. Il fait lui-même parti du projet pour *Rigorous Engineering of Mainstream Systems* initié par l'université de Cambridge visant à fournir des outils d'analyse et de preuve de programme afin de fournir de meilleures propriétés de sûreté et de sécurité.

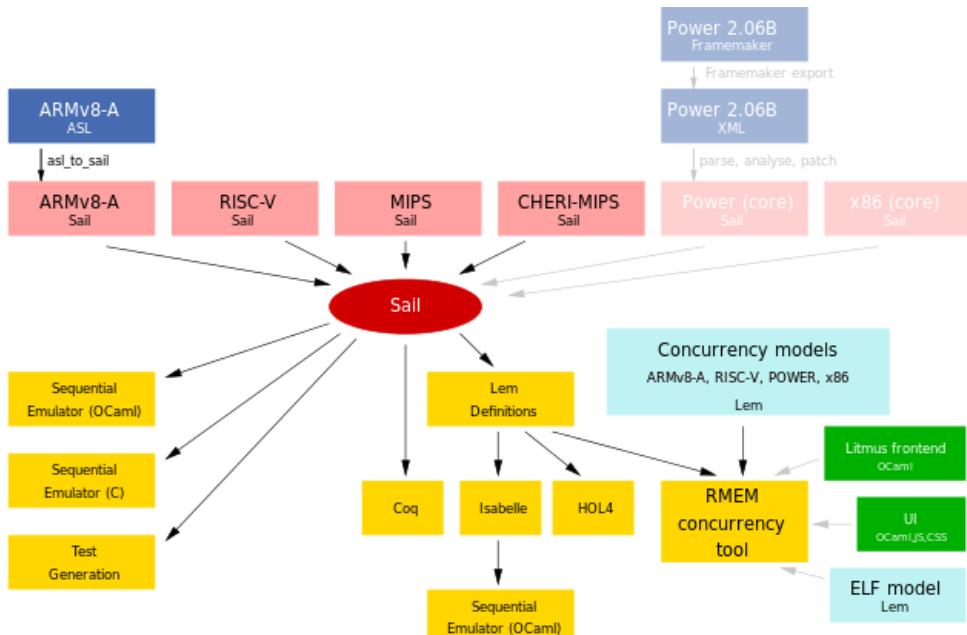


FIGURE C.1 – Architecture des composants de **Sail**

La Figure C.1 montre l'architecture et les différents composants de **Sail**. Bien qu'un intérêt tout particulier ait été donné à ARMv8-A **Sail** supporte d'autres architectures comme RISC-V, MIPS et x86 (de manière incomplète). La représentation intermédiaire s'appuie sur un type à base de types dépendants. Le **Sail** permet ensuite de générer

179. <https://github.com/rems-project/sail>

des émulateurs dans plusieurs langages dont C, OCaml ainsi que des définitions pour des preuves Isabelle<sup>180</sup>, HOL4<sup>181</sup> et Coq<sup>182</sup> (actuellement pour MIPS uniquement). Le support des architectures est suffisamment complet pour faire *booter* un noyau Linux<sup>183</sup>, FreeBSD ou encore un noyau seL4<sup>184</sup>. Cela implique naturellement le support des instructions privilégiés et tout ce que cela implique (translation d'adresses virtuelles en adresses physiques etc.).

**Support de ARM.** Ces travaux ont été fait en collaboration avec Alastair Reid travaillant chez ARM Ltd. À partir des versions pdf des manuels [240] il a d'abord été capable d'extraire le comportement des instructions spécifié dans un langage appelé ASL puis de les transformer en XML pour les réifier en **Sail**. Ce travail titanique est à la hauteur des résultats car le support est presque total. Il implique le support des instructions, des flottants, des *page table*, des interruptions, des exceptions, du mode hyperviseur et du *secure mode*. L'article scientifique décrit les deux jeux de sémantiques qui ont été créés :

- sémantique des instructions AArch64 de la spécification 8.3 en 23K lignes de **Sail** représentant 344 instructions (écrites en 1479 fonctions, et 245 registres **Sail**)
- sémantique des instructions AArch64 la spécification 8.3 issue de la documentation privé de ARM, implémentant 390 instructions (46 instructions système en plus)

Cette dernière étant issue d'une documentation interne à ARM, elle n'a pas été publiée. C'est sur celle-ci qu'il est possible de *booter* Linux grâce à la sémantique d'une centaines de registres systèmes dont certains sont nécessaires au *boot*. Celle-ci implémente aussi des sémantiques liées au *timers*, *memory-mapped I/O* (pour UART) et les *generic interrupt controller* (GIC).

---

180. <https://isabelle.in.tum.de/>

181. <https://hol-theorem-prover.org/>

182. <https://coq.inria.fr/>

183. pour ARM requiert la sémantique d'instructions issues d'une documentation interne à ARM

184. <https://sel4.systems/>

## Annexe D

---

### Différence approche CP et SMT

---

CP et SMT s'attaquent tous deux à la résolution de problème d'optimisations via des approches différentes et historiquement appliqués à des domaines de recherche différents. Le problème le plus courant est SAT où l'on cherche une solution à une formule booléenne (donc un valuation de chaque atome) mais l'on peut aussi vouloir trouver toutes les solutions d'un problème ou une solution optimale selon certains critères (typiquement minimisation ou maximisation d'une variable). En fonction du problème à résoudre l'approche et donc le solveur à utiliser seront potentiellement CP ou SMT[301].

La programmation par contrainte *Constraint Programming* est historiquement utilisée en [Intelligence Artificielle \(IA\)](#) pour des problèmes de combinatoires où les variables peuvent prendre plusieurs valeurs (autrement dit non booléennes). De par leur utilisation en [IA](#), les solveurs CP utilisent des stratégies et des méthodes de propagation dédiées à l'[IA](#). Par exemple la stratégies `all-different`<sup>185</sup> cherche à donner une valeur différente à toutes les variables de la formule. Les valeurs de variables sont généralement représentée sous forme de domaines (dédiée au problème à résoudre). Le solveur va essayer d'éliminer des valeurs possibles en explorant les différentes clauses de la formule après avoir assigné des valeurs “candidates” et en revenant en arrière si l'assignation trouve un contre-exemple.

Les solveurs [SMT](#) sont historiquement utilisés dans la communauté de la vérification. Un problème de *Satisfiabilité Modulo des Théories* (SMT) peut s'encoder en un problème de *Satisfiabilité* (SAT) qui fournit des informations sur des variables booléennes dont les clauses sont représentées en forme conjonctive. Le SMT fournit un certains nombres de théories où chaque théorie représente un nombre potentiellement infini de clauses SAT. De par son utilisation en vérification la majorité des théories représentent des structures rencontrées dans les programmes (Entiers, tableaux, etc). Ainsi un SMT est une généralisation d'instance de SAT dans lequel les variables ont été remplacées par des prédicats dans diverses théories.

L'évolution des solveurs SMT a réduit les différences notables entre les deux approches. Notamment certains solveurs supportent le symbole `distinct` qui permet d'ob-

---

185. <http://web.imt-atlantique.fr/x-info/sdemasse/gccat/Calldifferent.html>

#### *Annexe D. Différence approche CP et SMT*

---

tenir deux valeurs distinctes pour deux variables ou encore d'autres implémentent la stratégie `all-different` [20]. Néanmoins des différences persistent ; les solveurs CP sont conçus pour traiter des problèmes sur des domaines finis alors que les SMT supportent les domaines infinis (entiers et réels). En soi, SAT est un problème CP sur le domaine des booléens mais le fonctionnement des solveurs SAT en termes de propagation et d'apprentissage de clauses diffèrent du CP. Néanmoins certains solveurs CP traduisent leurs problèmes en instance SAT. Aussi bien CP que SMT permettent donc d'exprimer des instances du problème SAT en utilisant une expression du problème beaucoup plus naturelle qu'en écrivant la formule SAT à la main. Ainsi, bien que les approches CP et SMT soient différentes dans leur fonctionnement interne les différences en termes de fonctionnalités tendent à s'uniformiser (avec en tête les solveurs SMT de plus en plus performants).

## Annexe E

---

# Résolution des débordements de buffer par angr

---

Cette annexe vise à expliquer les résultats anormalement faibles de `angr` sur les débordements de tampons dans les tests atomiques c'est à dire `CIId_55` (`bof_sample1`), `CIId_56` (`bof_sample2`) et `CIId_57` (`stack_bo_11`).

Il a été possible d'écrire un script (ci dessous 29) permettant de résoudre ces trois cas spécifiques. Ceux-ci nécessitait l'utilisation du mode d'exécution *explore* à la place de *run* (cf. 7.5.4). Cette méthode nécessite de passer l'adresse (ou l'état) à trouver. Il est aussi possible de spécifier une adresse à éviter. Il a donc été nécessaire de désassembler les trois programmes pour trouver les adresses cibles. De plus, il fut nécessaire de spécifier la taille des entrées, dans ce cas `angr` semble avoir des difficultés à identifier le caractère de terminaison null (0x00) d'une chaîne de caractères.

En particulier pour `CIId_55` le script ne résout le challenge que lorsque la longueur des entrées fournies est de 46. `CIId_56` fonctionne sans changer la taille de l'entrée (128) tandis que `CIId_57` nécessite une longueur  $8 < len \leq 12$ .

Cette approche est curieuse, car en essence, elle ne diffère que très peu du script original. En principe il aurait dû fonctionner sur ces trois tests (*bien que ce ne fut pas le cas*).

**Listing 29** Script de résolution des débordement de tampon pour angr

```
1 def solve(filename, input_length):
2     proj = angr.Project(filename)
3
4     func_addr = proj.loader.find_symbol('entry').rebased_addr
5     func_arg = claripy.BVS("input_string", input_length * 8)
6
7     start_state = proj.factory.call_state(func_addr, func_arg)
8
9     simgr = proj.factory.simulation_manager(start_state)
10
11    simgr.explore(find=TRIGGER_BB_ADDR, avoid=NOT_TRIGGER_BB_ADDR)
12
13    results = []
14    for state in simgr.found:
15        results.append(state.solver.eval(func_arg, cast_to=bytes))
16
17    return results
```

## Annexe F

---

# Différence analyse de teinte et slicing en avant

---

La différence entre slicing et analyse de teinte tient essentiellement aux concepts qui sont manipulés.

La première notion est la controlabilité des entrées. L'analyse de teinte telle qu'elle est utilisée dans l'immense majorité des cas sert à modéliser toutes les entrées non-déterministes contrôlables par l'utilisateur et qui sont potentiellement à risque. Le slicing dans sa définition est beaucoup plus large et permet de slicer n'importe quelle variable.

L'analyse de teinte est une analyse sémantique de flot de données avec un domaine booléen représentant la teinte. Selon le cas, la propagation de la dépendance de donnée en slicing est essentiellement syntaxique et se base sur les chaînes def-use. Par exemple, l'analyse de teinte ne propagera pas la teinte sur l'opération  $a = b \oplus b$ , alors qu'un slicing syntaxique propagera la dépendance.

Autre divergence, le slicing propage les dépendances de contrôle alors que l'analyse de teinte couramment utilisée en recherche de vulnérabilité est purement orientée donnée et se propage le long d'un seul chemin (parcouru par ailleurs par une autre analyse).

Malgré les quelques divergences de concept vu au dessus, le slicing en avant avec comme critère de slice l'instruction qui lis les entrées, est en essence, une analyse de teinte. L'analyse de teinte est donc une réification du slicing en avant ou le critère de slice référence nécessairement le/les *statement* dont les variables associées sont contrôlables par l'utilisateur.

*Annexe F. Différence analyse de teinte et slicing en avant*

---

# Métrique du nombre d'exécution par seconde

---

Plusieurs critères entrent en jeu lors de la comparaison des outils étudiés lors de cet état de l'art. La métrique comparative d'exécution par seconde n'a pas été retenue comme critère d'évaluation fiable de la qualité de l'outil, et ne représente qu'une information supplémentaire, qui ne peut faire foi de comparaison. En effet, dans l'optique de la recherche de vulnérabilité, l'objectif principal est de trouver un crash. La manière d'y arriver peut différer suivant les outils. Par exemple, AFL utilise la puissance de calcul de la machine afin d'obtenir un nombre d'exécutions par seconde conséquent, et ainsi épuiser l'espace de recherche au plus vite par **force brute**. À l'inverse, un DSE effectue moins d'exécutions par seconde mais résoud un chemin précis en une exécution. L'exécution est nécessairement plus lente, mais peut théoriquement être aussi efficace en termes de découverte de bugs qu'un fuzzer standard.

A titre indicatif, la métrique d'exécution par seconde a pu être récupérée pour AFL, Angora et Honggfuzz ; ces derniers rendant cette métrique facilement accessible. Sur l'exécutable base64, sur une durée d'une minute, les résultats sont les suivants :

- AFL : **3001 exec/s** pour **0 bugs**
- Honggfuzz : **1005 exec/s** pour **26 bugs**
- Angora : **976 exec/s** pour **39 bugs**

En se référant uniquement à cette métrique, AFL semble le plus efficace des trois outils, pourtant sur le temps imparti, ce dernier n'a identifié aucun bug, tandis qu'Honggfuzz en a identifié 26 et Angora 39. Ces résultats montrent qu'une métrique purement basée sur le nombre d'exécution par seconde ne serait pas représentative de l'efficacité de l'outil.

Une métrique plus intéressante serait celle du *nombre de bugs découverts pendant un temps imparti*, cette dernière représente mieux l'efficacité de l'outil. Mais l'échantillonnage du temps est alors critique. En effet, au début de l'exécution le fuzzer trouvera plus de bugs qu'après plusieurs heures d'exécution car la courbe de découverte de chemins suit une courbe asymptotique. Tout comme le nombre d'exécution par secondes l'échantillonage pose aussi des problèmes de représentativité de la mesure.

## Résumé

Ce rapport présente un état de l'art des différentes techniques de fuzzing, d'exécution concolique, de slicing ainsi que des techniques de combinaisons. Ce travail se veut aussi complet et didactique que possible, pour faire ressortir les points forts et les points faibles des différents travaux de recherche analysés. Dans ce contexte, dix outils ont été sélectionnés afin d'être comparés. Une suite de test a été conçue pour l'occasion afin d'évaluer le comportement idiosyncratique et les performances de chaque outil. Ce travail préliminaire s'inscrit dans une logique d'élaboration et d'implémentation d'une analyse combinée ayant pour objectif la recherche automatisée de vulnérabilités, et plus particulièrement sur des logiciels réseaux dédiés aux systèmes embarqués fonctionnant sur l'architecture ARM.

**Mots-clés:** fuzzing, exécution symbolique, exécution concolique, slicing, combinaisons, vulnérabilités

