Découverte automatique de bugs au sein de binaires

Exécution symbolique dynamique et fuzzing

université de BORDEAUX

Mémoire de fin d'étude

Master *Sciences et Technologies*,
Mention *Informatique*,
Parcours *Cryptologie et Sécurité Informatique*.

Auteur
Florent Saudel <florent.saudel@etu.u-bordeaux.fr>

Superviseur
Georges Bossert <gbt@amossys.fr>
Frédéric Guihéry <fgy@amossys.fr>

Tuteur
Emmanuel Fleury <fleury@labri.fr>

23 octobre 2015

Déclaration de paternité du document

Je certifie sur l'honneur que ce document que je soumets pour évaluation afin d'obtenir le diplôme de Master en *Sciences et Technologies*, Mention *Mathématiques* ou *Informatique*, Parcours *Cryptologie et Sécurité Informatique*, est entièrement issu de mon propre travail, que j'ai porté une attention raisonnable afin de m'assurer que son contenu est original, et qu'il n'enfreint pas, à ma connaissance, les lois relatives à la propriété intellectuelle, ni ne contient de matériel emprunté à d'autres, du moins pas sans qu'il ne soit clairement identifié et cité au sein de mon document.

Date et Signature

Résumé

Face à la prolifération de logiciels, de systèmes d'informations et des réseaux les reliant, il apparaît obligatoire de s'assurer de leur sécurité. La recherche automatique de vulnérabilités devient alors nécessaire lorsque l'on considère l'ampleur de la tâche. Par ailleurs, certaines organisations s'y intéressent de très près comme le montre la compétition Cyber Grand Challenge [1] organisée par la DARPA et destinée à promouvoir le développement (américain) dans le domaine. Ce mémoire porte sur les techniques à l'état de l'art s'adressant au problème de la recherche automatique de vulnérabilités et plus particulièrement à celles dites dynamiques. Parmi elles, deux techniques semblent s'imposer : l'exécution symbolique et le fuzzing. Et sur ces deux techniques que le mémoire se concentre.

La première partie présente le fonctionnement d'un exécuteur dynamique symbolique et la théorie sous-jacente. La seconde, quant à elle, s'attarde sur les possibilités d'interactions entre un exécuteur symbolique et un fuzzer. Malgré leur approche très différente du problème, ils semblent y avoir une certain complémentarité entre les deux outils. Afin de confirmer cette intuition nourrie par les travaux de Majundar et Sen [35], Pak [39] ou encore Chen et al [18], nous présentons un prototype ainsi que les résultats obtenus.

1. `https://cgc.darpa.mil/`

Présentation de l'entreprise

Ce travail a été effectué au cours d'un stage de 6 mois au sein de la société AMOSSYS. La section qui suit fait le tour de ses activités et de son organisation.

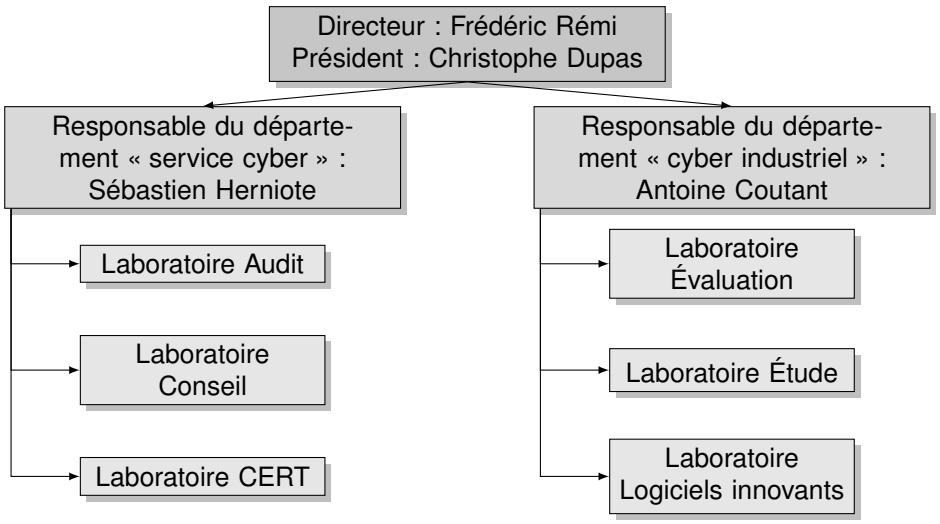
Activité

AMOSSYS [43] est une société indépendante de conseil et d'expertise en cybersécurité fondée en 2007. Sa direction est assurée par Frédéric Rémi et Christophe Dupas. Il s'agit de l'une des rares entreprises françaises agréée par l'ANSSI et accréditée par le COFRAC (Comité Français d'Accréditation) ayant pour but d'évaluer la sécurité d'un produit. Parmi les clients d'AMOSSYS, nous trouvons des administrations telles que des ministères ou des organismes interministériels mais aussi des entreprises publiques ou privées tels que de grands groupes industriels, des banques, PME, etc.

Cette société est en mesure d'intervenir sur l'ensemble du cycle de vie d'un projet, depuis les phases amont de recherche et de développement, de recueil, de besoin et de spécifications, jusqu'à celles qui se rattachent à la mise en œuvre et à la maintenance d'un produit ou d'un système. Les clients sont accompagnés dans la sécurisation et la défense de leurs systèmes d'information en cohérence avec leurs enjeux et leurs besoins. Basée à Rennes, cette entreprise jeune et dynamique compte une quarantaine d'experts répartis autour de six laboratoires : audit, conseil, CERT, évaluation, recherche & développement et logiciels innovants.

Organigramme

L'organigramme suivant présente la structure générale de l'entreprise. Celle-ci est divisée en deux départements qui sont séparés en laboratoires.



Audit

Le laboratoire audit a pour but d'auditer un système d'information ou un produit à un instant T, afin d'en ressortir ses points faibles et ses points forts. Le référentiel général de sécurité (RGS) est le cadre règlementaire permettant d'instaurer la confiance dans les échanges au sein de l'administration et avec les citoyens. Il propose une méthodologie de responsabilisation des autorités et des règles que doivent mettre en œuvre les administrations dans le cas de certifications ou d'audit de sécurité. Différents types d'audit peuvent être réalisés : audit intrusif, audit d'architecture, audit de configuration, audit d'applications et audit organisationnel. AMOSSYS est un Prestataires d'Audit de la Sécurité des Systèmes d'Information (PASSI) qualifié par l'ANSSI.

Conseil

Le laboratoire conseil a pour but de sensibiliser ses clients et de gérer les risques de la sécurité des systèmes d'informations en établissant des postures robustes face aux attaques informatiques, ce qui permet de se prémunir de celles-ci. Pour cela, des analyses de risques sont notamment effectuées afin d'identifier les points jugés critiques et établir une politique de sécurité.

CERT

Le laboratoire CERT (Computer Emergency Response Team) intervient suite à un incident de sécurité comme une compromission d'information. Une équipe est dépêchée sous 24h dans les locaux de l'entreprise en France métropolitaine afin d'investiguer pour être en mesure de réagir. Des collectes et des analyses d'informations sont réalisées, des analyses et des rétro-conceptions de codes malveillants peuvent être effectuées. Enfin, un durcissement du niveau de sécurité est mis en place. De plus, un travail de veille est réalisé afin de se tenir au courant des dernières découvertes technologiques d'une part, et de sensibiliser les clients d'autre part.

Étude

L'objectif principal de ce laboratoire est de produire l'architecture et les spécifications des projets clients. Des analyses de vulnérabilités de produits au sein d'une architecture sont réalisées ce qui permet de produire une synthèse personnalisée pour chaque client. En plus de l'aspect offensif, un aspect défensif est aussi étudié. En effet, des mécanismes de détection sont mis en œuvre en fonction du besoin. Il peut s'agir de détecteurs d'intrusion, d'antivirus, de pare-feu, etc.

AMOSSYS participe également à plusieurs projets de recherche portant sur des preuves de concepts ou encore des études en cybersécurité. Ce laboratoire participe à des conférences et encadre des thèses. Par exemple :

- *l'identification des algorithmes cryptographiques dans les produits de sécurité* : depuis octobre 2013, cette thèse met en relation les domaines de l'analyse de binaires et de la cryptographie. Elle vise à faciliter l'identification automatisée des algorithmes cryptographiques implémentés dans les produits de sécurité ou dans les malwares. Elle a également pour but de contribuer à l'état de l'art en permettant la détection d'algorithmes inconnus ou présentant des variations significatives vis-à-vis d'algorithmes connus. Un article est disponible à l'adresse suivante : `http://blog.amossys.fr/Automated%20Reverse%20Engineering%20of%20Cryptographic%20Algorithms.html`.

Il a également pour but de développer des outils pour améliorer les analyses des autres laboratoires. Parmi ces outils nous pouvons citer :

- **Netzob** : (NETwork protocol modeliZatiOn By reverse engineering) un logiciel libre de rétroingénierie, de génération de trafic et de fuzzing des protocoles de communication, mis au point dans la cadre de la thèse de Georges Bossert : « Exploiting semantic for the automatic reverse engineering of communication protocols ». Il est disponible à l'adresse suivante : `http://www.netzob.org/` et a été présenté lors de différentes conférences : CCC, Black Hat, ICCC, SSTIC, SARSSI ;
- **Hooker** : une solution automatisée d'analyse de markets Android disponible à l'adresse suivante : `https://github.com/AndroidHooker/hooker`. Cet outil a été présenté lors de la conférence du SSTIC 2014 ;
- **PyCAF** : un framework Python destiné à aider les auditeurs lors de l'analyse de la configuration des équipements d'un système d'information. Cet outil a été présenté lors de la conférence du SSTIC 2015 et est disponible à l'adresse : `https://github.com/maximeolivier/pyCAF`.

Évaluation

Ce laboratoire a pour principal objectif d'évaluer la conformité et l'efficacité des produits du type « logiciels et équipements de réseaux ». AMOSSYS est un CESTI (Centre d'Évaluation de la Sécurité des Technologies de l'Information) agréé par l'ANSSI. Ces évaluations sont réalisées dans le cadre de la CSPN (Certification Sécurité de Premier Niveau) et selon les CC (Critères Communs).

Une évaluation CSPN (schéma promu par le centre de certification français, l'ANSSI) a pour but d'assurer que les services de sécurité annoncés par une cible d'évaluation soient effectivement rendus. La philosophie de la CSPN est d'obtenir des résultats fiables en temps contraint (25 hommes.jour et 10 hommes.jour pour l'expertise cryptographique). Les analyses menées visent à s'assurer de la robustesse d'une cible d'évaluation. La mise en place de ce schéma d'évaluation (2008 à 2011 pour la phase expérimentale) est issue du constat suivant :

- offre assez limitée de produits SSI dont la qualité a été attestée ;
- absence de certification portant sur des logiciels libres ;
- coûts et délais d'évaluation peu ou pas adaptés au marché des produits SSI civils ;
- produits de sécurité avec des mécanismes de base de plus en plus complexes et hétérogènes, souvent hors de portée du cadre normatif et de plus en plus souvent intégrés au système d'exploitation ou à l'environnement opérationnel.

La mise en place du schéma CSPN répond donc à des besoins liés au marché de l'évaluation de produits de sécurité civils (temps contraint adapté au versionnage des produits, coûts optimisés) et répond à des besoins purement techniques tels que privilégier l'expertise technique à la complétude méthodologique.

Les Critères Communs (norme ISO 15408), désignés aussi par le terme CC ou Common Criteria, sont une méthodologie d'évaluation de la sécurité dans le domaine des technologies de l'information. C'est une méthodologie ouverte, publique, gratuite et reconnue internationalement. De ce fait, les CC sont utilisés partout dans le monde et, en vertu d'accords internationaux, les certificats émis par une autorité nationale peuvent être reconnus hors du pays de certification. Dans le cadre des Critères Communs, l'évaluation d'un produit restreint à un périmètre appelé cible d'évaluation se subdivise en deux sous-ensembles complémentaires :

- l'analyse des documents de développement (et du code source selon le niveau d'assurance visé) sous l'angle de la conformité ;
- le test du produit et la recherche de vulnérabilités sous l'angle de l'efficacité (avec un potentiel d'attaquant augmentant avec le niveau d'assurance).

AMOSSYS est également reconnu par l'Autorité de Régulation des Jeux En Ligne (ARJEL) comme organisme certificateur, autorisé à accomplir les opérations de certification des opérateurs de jeux d'argent et de hasard en ligne.

Logiciels innovants

Ce nouveau laboratoire, créé en janvier 2015, a pour vocation le développement de produits liés à la sécurité. Composé de développeurs multi-compétences, ce laboratoire intervient sur toutes les phases d'un projet : de la création de l'algorithme à l'implémentation de l'interface graphique. Les travaux de ce laboratoire sont généralement réalisés suite à des appels d'offre.

Table des matières

Conclusion

Annexes

Première partie

Etat de l'art et aspect théorique

Connu depuis 40 ans, l'exécution symbolique est une technique d'analyse de programme qui connaît un regain d'intérêt auprès des communautés liées à la sécurité informatique et à la vérification logicielle. Les deux raisons principales de ce renouveau est l'augmentation des capacités (mémoire et puissance de calcul) des ordinateurs actuelles ainsi que l'amélioration des *theorem provers*. Ces deux outils jouent, en effet, un rôle important dans l'efficacité d'une exécution symbolique. Son champ d'application est large et comporte entre autre la génération de tests automatiques [41, 14, 45], la génération de condition de vérification (VC) [4], l'analyse de *malware* [19, 44] ou encore la recherche de bugs et de vulnérabilités [3, 49].

Le but de cette partie est d'apporter une vision générale sur l'exécution symbolique, son fonctionnement, ses limites et les pistes intéressantes pour de futures améliorations. Nous commençons par une description générale (chaptire 1) de l'exécution symbolique, puis nous détaillons les avantages et inconvénients (chapitre 1.4) selon le contexte d'exécution : statique ou dynamique. Par la suite, nous nous concentrons principalement sur l'exécution symbolique dans un contexte dynamique (chapitre 3). Nous présentons ensuite les pistes d'optimisations connues pour mitiger l'explosion combinatoire (chapitre 3.3) du nombre de chemin par rapport à la taille du programme analysé, aussi connu sous le nom de *path explosion problem* [19]. Enfin, le dernier chapitre expose les différents composants logiciels d'un exécuteur symbolique dynamique de binaire (chapitre 3.4).

CHAPITRE 1

Exécution symbolique

L'exécution symbolique permet de construire des formules relatives au déroulement d'un programme. Selon la manière dont est effectuée l'exécution symbolique, il est possible de récupérer des informations sur le flot de contrôle [26] comme sur le flot de donnée [4] en résolvant ces formules.

Les paragraphes suivant expliquent le fonctionnement général de l'exécution symbolique et introduisent un ensemble de notions qui lui sont propres.

1.1 Description

Afin de réaliser l'exécution symbolique d'un programme, les entrées de celui-ci doivent être remplacées par des symboles algébriques. Au lieu de fournir des nombres ou une chaîne de caractère, les entrées du programme sont des symboles libres, capable de prendre n'importe qu'elle valeur. A partir de là, l'exécution se déroule dans le domaine algébrique. Chaque instruction ou opération se voit remplacer par son penchant algébrique qui renvoie une formule. Pour les deux instructions suivantes, en pseudo-langage C [1] :

```
1    x = x + y;
2    y = x*10 - 5*y;
```

Si `x` et `y` se voient affectés respectivement α et β dans le domaine algébrique. Le résultat à l'issu de l'exécution symbolique de cet exemple sont les formules suivantes : $\alpha + \beta$ pour `x` et $(\alpha + \beta) \times 10 - 5 \times \beta$ pour `y`. Le traitement est similaire pour tout code linéaire un flux d'instruction linéaire.

Les instructions de branchement (`if, while` ...) doivent être traitées de manière particulières. En effet, les expressions symboliques représentent un ensemble de valeurs, par conséquent il est impossible de connaître la branche prise sans s'intéresser à la formule associée à la condition. Nous allons donc voir comment les instructions de branchement sont gérées à travers un exemple

1. Dans la suite, l'ensemble des exemples de code donnés seront écris en C, cela dans un soucis de simplicité. Les principes énoncés ne se limitent en aucun cas au programme écrit dans ce langage.

de programme (Listing 1.1) et définir de façon plus formelle le déroulement d'une exécution symbolique.

L'exécution symbolique se réalise au travers d'une machine à état symbolique [48, 44]. Elle se compose de 3 éléments :

- un pointeur d'instruction (`ip`) qui pointe vers l'instruction courante.
- un vecteur symbolique (vs) qui associe à chaque variable du programme sa valeur symbolique. Par exemple $(X \rightarrow \alpha, Y \rightarrow \beta, Z \rightarrow 3)$. La valeur symbolique d'une variable peut être une constante comme le montre la variable Z. Sa longueur varie en fonction des nouvelles variables rencontrées et lors des sorties des blocs.
- un *path condition* (`pc`). Son rôle est d'agréger au fur et à mesure de l'exécution les contraintes sur le flot de contrôle. A la fin, le *path condition* est une formule issue de la conjonction de toutes les contraintes qui caractérisent l'exécution réalisée par la machine à état. L'ensemble des solutions de cette formule (appelé aussi le modèle) est l'ensemble des valeurs des entrées (ici X et Y) pour lesquelles le flot de contrôle du programme suivra exactement la même séquence d'exécution.

La composition exacte d'une telle machine symbolique dépend de la nature du langage et de son niveau d'abstraction. Par exemple pour un langage plus bas niveau tel qu'un assembleur, la machine devra comprendre un modèle de mémoire capable de stocker des valeurs symboliques [3] Ici notre version simplifiée est suffisante pour les exemples à venir.

1.2 Définitions et exemple

Dans cette section, nous utilisons la fonction *scoring* (Listing 1.1) en tant qu'exemple afin d'introduire des notions propre à l'exécution symbolique et montrer de manière plus détaillée le déroulement d'une exécution symbolique.

La figure 1.1 représente l'arbre issue de l'exécution symbolique de la fonction *scoring*. Chaque nœud de l'arbre représente un état de la machine symbolique [2]. Chaque chemin de cet arbre, appelé arbre d'exécution [32], représente une classe de traces d'exécution de la fonction. La formule (*path condition*) correspondant à ce chemin est donnée au niveau de sa feuille. Tous les appels à la fonction *scoring* avec un paramètre `x` satisfaisant cette formule suit obligatoirement ce chemin au sein de l'arbre. La trace d'exécution appartient alors à la classe correspondante au chemin en question et l'entrée fournie à la fonction, la valeur actuelle de `x`, appartient au modèle du *path condition*.

L'exécution symbolique à l'origine de cet arbre d'exécution est présentée par la suite. A l'initialisation, la machine à état symbolique pointe sur la première instruction de la fonction (cf. ligne 2). Le vecteur symbolique est $(x \rightarrow \alpha, y \rightarrow 0)$ puisque y est initialisé à 0 et x contient la variable symbolique α car en tant qu'argument de la fonction sa valeur n'est pas définie. Enfin, le *path condition* est initialisé à $true$ car il n'y a pour l'instant aucune contrainte

2. Par soucis d'espace, certains états ont été fusionnées. Par exemple les instructions ligne 2 et 4 partagent le même état à l'exception du pointeur d'instruction (nœud 1). Pour les mêmes raisons, la valeur symbolique associé à x n'est pas rappelée au sein des différents états car celle-ci ne change pas. Le champ **id** n'est renseigné que par soucis de clarté, il n'intervient pas au cours de l'exécution symbolique.

Listing 1.1: Fonction scoring

```
1  int scoring(int x) {
2     int y = 0;
3
4     if (x < 0)
5        y = -x + 1;
6
7     if (x > 15)
8        y += 2;
9
10    return y;
11 }
```

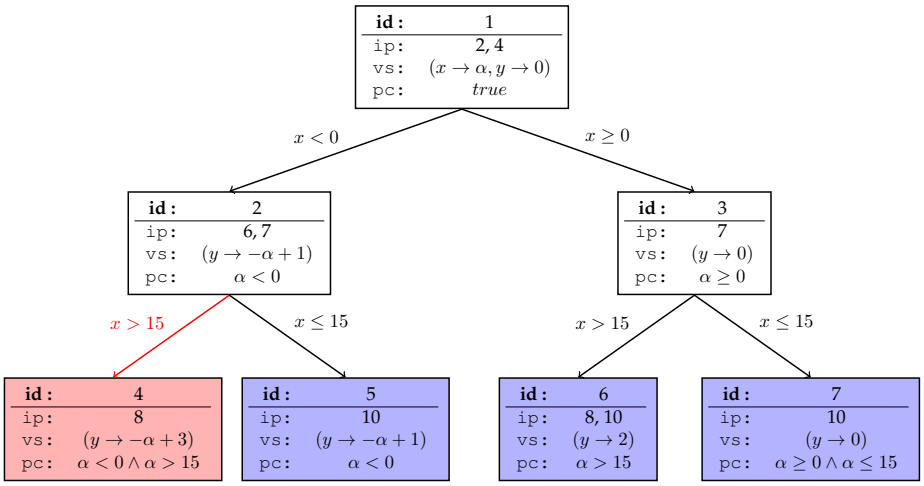


FIGURE 1.1: Arbre d'exécution de la fonction *scoring* (Listing 1.1)

sur le flot de contrôle. Cet état initial est représente par la racine de l'arbre nœud 0. A la ligne 4, une première instruction de branchement est exécutée. A partir de là, l'exécution du programme peut prendre deux directions : la première où $x < 0$ et la seconde où $x \geq 0$. Afin de prendre en compte ces deux possibilités, la machine à état symbolique est dupliquée (*fork*), la suite de l'exécution symbolique se fera le long des deux machines à état (nœud 2 et 3).

Chaque machine se voit mettre à jour son `pc` de la manière suivante. Pour une condition c, le nouveau *path condition* est égal à la conjonction de l'ancien `pc` et de la condition : $\text{pc}_{new} \leftarrow \text{pc}_{old} \wedge c$.

Nous faisons le choix arbitraire de commencer par la branche $x < 0$, l'ordre ici importe peu. L'instruction suivante (cf. ligne 6) modifie la valeur de y en lui affectant la formule algébrique $-\alpha + 1$. Vient ensuite le branchement conditionnel ligne 7, dont la condition est $x > 15$. Normalement, la machine symbolique devrait donc être encore une fois dupliquée. Le premier état aurait donc un `pc` valant $\alpha < 0 \wedge \alpha > 15$ (nœud 4). Or cette formule ne peut

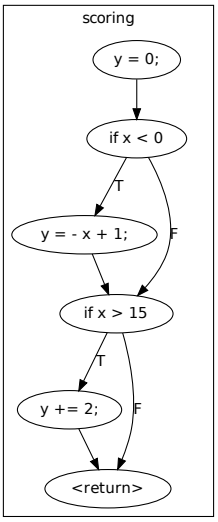


FIGURE 1.2: Graphe de flot de contrôle (CFG) de la fonction *scoring* (Listing 1.1)

être satisfaite, elle est évidemment tout le temps fausse. Par conséquent il est impossible que le flux d'exécution du programme puisse emprunter un chemin correspondant à ce `pc`. Aucune duplication n'est alors faite et l'exécution suit l'autre branche (nœud 5) qui, quant à elle, possède un `pc` satisfaisable $(\alpha < 0 \wedge \alpha \leq 15) = \alpha < 0$. Par ailleurs, la dernière instruction de la fonction *scoring* est atteinte. L'exécution symbolique s'arrête donc pour ce chemin. Le reste de l'exécution symbolique se poursuit à partir du nœud 3 en suivant un processus similaire jusqu'à atteindre toutes les fins possibles de la fonction *scoring* (nœud 6 et 7).

Maintenant que le concept d'exécution symbolique est introduit de façon informel, nous allons nous intéresser à l'algorithme sous-jacent.

1.3 Algorithme

L'algorithme 1 présente une implémentation haut niveau de l'exécution symbolique telle que nous l'avons utilisée pour traiter l'exemple de la fonction *scoring*. Celui-ci repose sur l'utilisation d'une *worklist* contenant les *états* restant à parcourir.

A chaque tour de boucle, un *état* est récupéré dans la *worklist*, l'instruction sur laquelle l'*état* pointe est récupérée avant d'être exécutée symboliquement. A chaque type d'instruction est réservé un traitement particulier correspondant à sa sémantique : un saut conditionnel et une affectation non pas le même effet sur un *état*. L'algorithme 1 présente une vision simplifiée du traitement de trois type d'instructions courantes. L'exécution symbolique pour une affectation est simple. L'algorithme crée un nouvel état avec un `ip` pointant vers la

prochaine instruction (`succ`) et un vecteur symbolique mis à jour afin que la variable v soit associée au résultat de l'expression e. Lors d'un branchement, l'algorithme vérifie si les chemins peuvent être réellement exercés, c'est-à-dire que le `pc` est satisfaisable. Si c'est le cas, un nouvel *état* est créé avec un nouveau `pc` mis à jour : $pc \wedge \neg c$ ou $pc \wedge c$. Si l'instruction signe la fin de l'exécution en cours (*halt*), l'algorithme affiche simplement le *path condition* du chemin. L'ordre dans lequel les *états* sont exécutés dépend entièrement du choix effectué par la fonction `pickNext`.

Nous allons maintenant nous intéresser aux points forts de l'exécution symbolique.

Algorithme 1 Algorithme générique d'exécution symbolique [3, 33]

$\mathcal{W} \leftarrow \{(\text{ip}_0, \emptyset, \text{true})\}$ ▷ (`ip`, `vs`, `pc`)
while $\mathcal{W} \neq \emptyset$ **do**
  $((ip, vs, pc), \mathcal{W}) \leftarrow$ `pickNext`$(\mathcal{W})$
  $S \leftarrow \emptyset$
  switch `ins`(ip, vs) **do**
    case $v := e$ ▷ Affectation
      $S \leftarrow S \cup \{(\texttt{succ}(ip), vs[v \rightarrow \texttt{eval}(e, vs)], pc)\}$
    case if c **goto** ip' ▷ Branchement
      if `isSAT`$(pc \wedge c)$ **then**
        $S \leftarrow S \cup \{(ip', vs, pc \wedge c)\}$
      else if `isSAT`$(pc \wedge \neg c)$ **then**
        $S \leftarrow S \cup \{(\texttt{succ}(ip), vs, pc \wedge \neg c)\}$
      end if
    case $halt$ ▷ Arrêt
      `print`(pc)
  end switch
  $\mathcal{W} \leftarrow \mathcal{W} \cup S$
end while

1.4 Avantages

L'exécution symbolique permet de construire l'arbre d'exécution d'un programme. Si celle-ci s'achève après avoir traversée l'ensemble des chemins réalisables, l'exécution symbolique aura effectué une couverture totale du code [32, 28]. Pour chaque chemin p est alors associé une formule ϕ_p. Tous les ϕ_p sont satisfaisables car elles correspondent aux chemins réalisables concrètement. Les solutions d'une formule ϕ_p correspondent aux entrées du programme exerçant le chemin p. Avec cette définition, l'exécution symbolique a une précision « parfaite » [26]. Elle exerce donc la totalité du programme sans aucune approximation.

Ceci tient en théorie, cependant l'exécution symbolique rencontrent plusieurs difficultés en pratique.

1.5 Limites

A l'instar des autres analyses applicables sur un programme, il est possible de réaliser une exécution symbolique statiquement ou dynamiquement. Quelque soit son type, une exécution symbolique fait face à différents challenges :

- une explosion combinatoire sur le nombre d'états.
- les algorithmes cryptographiques, les fonctions de hachages et autres fonctions dîtes complexes.
- l'arithmétique des pointeurs symboliques.
- les opérations sur les nombres flottants.
- l'interaction avec l'environnement extérieur.

Explosion combinatoire ou *path explosion problem*

La première limite de l'exécution symbolique vient de l'arbre d'exécution. Même un programme simple peut posséder un nombre de chemin infini et par conséquent un arbre d'exécution infini, pour s'en convaincre prenons l'exemple d'un petit programme (Listing 1.2) acceptant des entrées de taille non bornée. En pratique, même s'il n'est pas infini l'arbre d'exécution est exponentiellement plus grand que le nombre d'instructions au sein du programme [19, 44, 41], ce qui est à l'origine du *path explosion problem*.

Listing 1.2: Programme ayant un arbre d'exécution infini

```
1 int main(int argc, char **argv) {
2    if (argc != 2)
3       return -1;
4
5    unsigned int i = 0;
6    while(argv[1][i] != '\0')
7       i++;
8
9    return i;
10 }
```

Ce problème a plusieurs implications en pratique. L'arbre d'exécution étant trop important l'exécution symbolique requiert de nombreuses ressources en temps et/ou en mémoire. Par ailleurs, l'exécution symbolique sur des exemples non triviaux a peu de chances de se terminer dans un temps raisonnable.

Limites du SMT solver : fonctions complexes

A plusieurs reprises au cours de l'exécution de la fonction *scoring* (Figure 1.1), il a fallut décider si des formules étaient satisfaisables ou non. Pour répondre à cette question, l'exécution symbolique repose sur un *theorem prover* ou *SMT solver*. Or ces outils sont eux-mêmes limitées de part la nature du problème qu'il traite. Nous parlons du problème SMT de façon plus détaillé par la suite au sein du chapitre 3.4.

Les limites des SMT solver actuels se répercutent sur celles de l'exécution symbolique. Par exemple, si un chemin possède un *path condition* hors de portée des SMT solver actuels, l'exécution symbolique se trouvera face à un

dilemme. Soit, elle considère le chemin comme réalisable. Au quel cas s'il ne l'est pas, l'exécution couvre des chemins inexistants et peut donc produire des faux positifs. Au contraire, si elle abandonne le chemin alors que celui-ci est réalisable alors des bouts de l'arbre d'exécution ne sont pas traités (faux négatifs). L'exécution symbolique n'est donc pas complète. Un test sur la valeur d'un haché ou sur le résultat d'un algorithme cryptographique provoquera à coup sur ce genre de situation, le problème étant évidemment hors de portée des SMT solvers.

Arithmétique des pointeurs

Les opérations sur les pointeurs symboliques soulèvent un autre type de problème. La question est de savoir ce que doit retourner le déréférencement d'un pointeur dont la valeur dépend en parti ou entièrement d'une expression symbolique. Comme par exemple `buf[i]` et `*(ptr + i)` dans le cas où l'expression symbolique associé à la variable `i` n'est pas une constante. Dans ce cas là, la pointeur résultant de l'évaluation de ces expressions C peut avoir de multiples valeurs. Dans le pire des cas, la variable `i` peut prendre n'importe quelle valeur. Cela signifierait que tous les octets de la mémoire pourrait être lus. La solution naïve consistant à considérer toutes les solutions de l'expression symbolique et créer autant de nouveaux états que nécessaire est donc potentiellement irréalisable. Afin de ne pas rester bloquer, une autre solution consiste à ne considérer qu'un sous-ensemble des valeurs possibles de `i` nous empêchant par la même d'exercer une partie de l'arbre d'exécution.

Ce problème est donc une autre source d'explosion combinatoire ou d'imprécision.

Les nombres flottants

Actuellement, la quasi-totalité des exécuteurs symboliques ne traitent pas les opérations sur les nombres flottants [22, 42]. Les effets sur l'état de la machine virtuelle sont perdu et aucune information sur le flot de contrôle ou le flot de données n'est récupérée. Cela peut donc être à l'origine d'imprécisions.

La raison principale de cette limitation est l'absence de SMT solvers robustes capables de raisonner à propos de la théorie des nombres flottants [19]. La recherche pour cette théorie est en retard par rapport au autres. L'article de Bagnara [6] datant de 2014 évoque un SMT solver FPSE implémentant la théorie FPA (*Floating Point Arithmetic*). Pour palier ce manque, d'autres outils comme Klee-mc [42](2014) propose de remplacer les opérations sur les flottants par leur équivalent basé sur des nombres entiers à l'aide de bibliothèque dédié. Ces approches semblent prometteuses mais encore très récentes et peu communes.

La gestion de l'environnement

Les logiciels d'aujourd'hui interagissent souvent avec l'environnement tel que le système d'exploitation, une bibliothèque partagée, le réseau ou l'utilisateur. De ce fait, l'exécution concrète du programme est influencée par d'autres événements extérieurs en plus des entrées de l'utilisateur. L'exécution symbolique doit donc les prendre en compte pour réaliser une analyse la plus

complète possible. Cependant, le programme ne voit ces interactions qu'à travers leur résultat. En effet dans la plupart des cas, il est impossible pour l'exécution symbolique de se poursuivre au sein de l'environnement et d'obtenir les contraintes liées à celui-ci. Pour palier ce problème, Klee [14] propose l'emploi de modèles simulant l'environnement. Chaque interaction avec l'extérieur doit être modélisée afin de fournir les contraintes correspondantes à l'ensemble des valeurs pouvant être retournées. Cette tâche réalisée manuellement est fastidieuse et propice aux erreurs.

L'exécution symbolique rencontre d'autres difficultés mais celle-ci sont spécifiques aux contextes dans laquelle elle s'effectue et sont traités dans les parties dédiées à chaque type d'exécution.

CHAPITRE 2

Exécution symbolique statique (SSE)

2.1 Une analyse statique

Tout au long de ce chapitre, nous détaillons l'exécution symbolique dans un contexte statique. Historiquement, elle correspond à la première version de l'exécution symbolique telle qu'elle a été présentée par J. King [32] en 1976. Une exécution symbolique statique (SSE) s'exécute en « lisant » le code. A aucun moment, le programme n'a besoin d'être exécuté (ni même compilé d'ailleurs). L'analyse s'effectue sur une représentation intermédiaire, un graphe de flot de contrôle par exemple, issue du *parsing* du code source. C'est exactement ce que nous faisons dans le chapitre 1.

Depuis, la SSE a été utilisée à des fins de vérification de logicielles ce qui a abouti à des améliorations. Nous présentons cela dans la section suivante. Par la suite, nous nous penchons sur ses limites avant de faire une aparté à propos de l'exécution symbolique statique d'un binaire.

2.2 Son utilisation au sein de la vérification logicielle

Dès les années 70, King et Hantler proposent l'utilisation de l'exécution symbolique afin de prouver la justesse d'un programme donné [32, 29, 48] au sein d'une analyse statique. A cette même époque, King évoque la possibilité d'ajouter des pré-conditions et des invariants de boucles au sein de l'analyse permettant ainsi d'utiliser la méthode de vérification de Floyd/Hoare [40] de façon automatisée. Depuis cette idée a été étendue à différentes approches de la vérification de programme [26].

Nous nous attardons sur une utilisation de l'exécution symbolique en particulier : la génération de conditions de vérification ou *verification condition* (VC). Nous expliquons, dans un premier temps, comment les *verification conditions* peuvent être utilisées pour s'assurer de la justesse d'un programme. Puis

nous abordons le rôle de l'exécution symbolique dans leur génération automatique.

A chaque instruction ou bloc d'instruction du programme est associé une *verification condition*. Une VC peut être représentée comme une formule ou un triplet de Hoare : une pré-condition P, une instruction S et une post-condition Q. Afin que le triplet de Hoare assure la justesse de S vis à vis de P et Q, P doit être vraie avant d'exécuter S et Q doit être vraie après son exécution. Cette même VC peut être représentée à l'aide de la formule suivante $P \wedge E_S \Rightarrow Q$ où E_S correspond aux effets du bloc d'instruction S. Si la formule est valide (tout le temps vrai) alors le bloc d'instruction est juste. En pratique, la vérification se déroule sur un programme contenant des assertions. La preuve consiste donc à s'assurer que pour tous les chemins au sein du programme et pour toutes les assertions au sein du programme, l'assertion précédente implique celle suivante (VC) en prenant en compte les effets des instructions [48]. Cela permet ainsi de vérifier automatiquement que le programme en question respecte la spécification prévue par les développeurs et qui est représentée au sein du code par les assertions.

Plus récemment des outils comme Saturn [50] et Calysto [5] ont fait l'usage de l'exécution symbolique à cette même fin. L'exécution symbolique joue le rôle ici de générateur automatique de VC. Elle construit une définition symbolique pour chaque variable du programme en tout point du programme. La conjonction de ces définitions pour un bloc d'instruction encadré par des assertions est la formule E_S. Il ne reste alors plus qu'à vérifier la validité de chaque VC automatiquement généré. Comme il est précisé dans la thèse de D. Babić [4], la génération de ces VC chemin après chemin comme nous le faisions jusqu'alors dans nos exemples est inefficace. Dans cette même thèse, Babić propose un algorithme modifiant l'exécution symbolique afin de prendre en compte tous les chemins possibles au moment de l'exécution. Cette forme d'exécution symbolique est connue sous le nom de *all-path forward symbolic execution* [24] ou encore de *static state merging* [33].

Pour présenter cette forme d'exécution, nous nous basons sur un exemple (Listing 2.1) extrait de la thèse de Babić [4] et son CFG (Figure 2.1). Nous calculons, à l'aide du CFG, la VC correspondant à ce bout de code et vérifions si l'exemple est juste. Pour ce faire, il est nécessaire de calculer la valeur symbolique de la variable $y1$ pour le nœud rouge où se situe l'assertion. Cette variable $y1$ est modifiée par les deux chemins partant de la racine du CFG jusqu'au nœud rouge. La définition que nous calculons doit donc prendre en compte ces deux chemins. Cela signifie que le nœud rouge doit être traité à la toute fin car il dépend des nœuds ($y1 = y0$) et ($y1 = y0 + x0$). Le parcours du CFG se fait donc en respectant un tri topologique[1]. L'évolution du vecteur symbolique au cours de l'exécution symbolique est présentée dans le tableau 2.1. Lorsque le moteur symbolique s'arrête sur le nœud rouge, la valeur de $y1$ dépend de la condition ($x0 < 0$). La variable vaut soit $y0$ ou soit $y0 + x0$. Cette idée est exprimée de manière plus concise avec l'opérateur ITE ou *if-then-else-operator*. La valeur symbolique à l'entrée du nœud rouge pour

1. Le tri topologique est un ordre de visite des sommets d'un graphe. Chaque sommet est traversé avant ses successeurs.

Listing 2.1: Exemple calcul de VC

```
1 x0 = a + b;
2
3 if (x0 < 0) {
4    x1 = -y0;
5    y1 = y0;
6 } else {
7    x1 = x0;
8    y1 = y0 + x0;
9 }
10
11 assert(0 <= y1);
```

$y1$ est donc

$$y1 = \text{ITE}(a + b < 0, y0, y0 + a + b)$$
$$\equiv$$
$$(a + b < 0 \Rightarrow y1 = y0) \vee (\neg(a + b < 0) \Rightarrow y1 = y0 + a + b)$$

Il faut maintenant calculer la *verification condition* :

$$(\mathit{true} \wedge y1 = \text{ITE}(a + b < 0, y0, y0 + a + b)) \Rightarrow (0 \leq y1)$$
$$\equiv$$
$$0 \leq \text{ITE}(a + b < 0, y0, y0 + a + b)$$

Pour terminer, il faut s'assurer que la formule $0 \leq \text{ITE}(a + b < 0, y0, y0 + a + b)$ est bien valide. Or, il s'avère que cette formule est fausse pour les valeurs suivantes : $a = 2$, $b = -3$ et $y0 = 0$. Donc le programme est faux aux yeux de cette assertion.

Les versions modernes du SSE sont donc capables de générés des formules encodant l'ensemble des chemins menant à un point du programme [3]. De ce fait, la SSE n'est pas confrontée directement au *path explosion problem*. Un parcours du CFG en suivant un tri topologique [5] permet de s'assurer que tous les chemins menant à un endroit du programme ont été visités avant lui. Le CFG étant fini et si nous le supposons acyclique, c'est-à-dire que le programme ne contient pas de boucle, l'exécution symbolique est sûre de se terminer [3]. Cependant la raison sous-jacente au *path explosion problem* n'a pas disparu. L'explosion combinatoire liée à la taille de l'arbre d'exécution se retrouve au sein même des formules : soit par leur taille ou soit par leur complexité [24].

2.3 Limites

La SSE a permis à des *static checkers* automatiques tels que Calysto [5] de trouver de nombreux bugs au sein de programme *open source* comme bind ou openldap. Calysto a détecté de nombreux bugs liés à des variables non initialisées ou des pointeurs nulles. Ce même principe peut être appliquer afin de

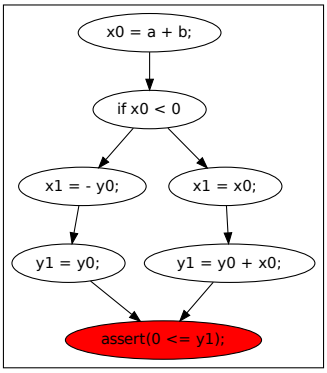


FIGURE 2.1: Graphe de flot de contrôle du Listing 2.1

Nœud	**Vecteur symbolique**
$x0 = a + b$	$\{x0 \to a + b\}$
if $(x0 < 0)$	$\{x0 \to a + b\}$
$x1 = -y0$	$\{x0 \to a + b,$ $x1 \to -y0\}$
$y1 = y0$	$\{x0 \to a + b,$ $x1 \to -y0,$ $y1 \to y0\}$
$x1 = x0$	$\{x0 \to a + b,$ $x1 \to x0\}$
$y1 = y0 + x0$	$\{x0 \to a + b,$ $x1 \to x0,$ $y1 \to y0 + x0\}$
assert$(0 <= y1)$	$\{x0 \to a + b,$ $x1 \to x0,$ $y1 \to \text{ITE}(x0 < 0, y0, y0 + x0)\}$

Tableau 2.1: Evolution du vecteur symbolique au cours de l'exécution symbolique du CFG (Figure 2.1)

détecter des vulnérabilités. Il suffit d'ajouter des assertions sur les bornes d'un tampon pour s'assurer qu'aucun débordement n'est possible. Néanmoins, la SSE possède certaines limites théoriques et pratiques fortes qui l'empêche de réaliser une analyse complète du programme.

La première difficulté de la SSE sont les boucles et les fonctions récursives. Par exemple, une boucle dont la condition contient des variables symboliques confronte la SSE au problème de l'arrêt. Autrement dit, savoir quand s'arrête une boucle ou une récursion est indécidable. En effet, chaque tour d'une boucle peut engendrer deux nouveaux chemins : une entrant dans la boucle et un sortant, à condition que les deux *path condition* correspondant soient satisfaisables. Dans le pire des cas, cela est vrai pour tous les tours. L'exécution symbolique « boucle » à son tour et ne se termine donc pas.

La SSE moderne délègue le *path explosion problem* au SMT solver en encodant de multiple chemins dans une formule unique. Cette formule est plus importante (taille) et plus complexe (présence de disjunction), ce qui rend la tâche plus difficile au SMT solver. Les limites propre au SMT solver sont donc plus rapidement atteinte.

Le contexte statique (code source du programme) manque d'un certains nombre d'informations. Toutes les interactions avec l'environnement externe sont autant d'inconnues pour la SSE. Par exemple, si le programme testé réalise un appel système ou à une fonction d'une bibliothèque partagée chargée dynamiquement, l'exécuteur symbolique est face à un dilemme. Soit, il traite, s'il est en capable, le code du noyau ou de la bibliothèque, ce qui alourdit énormément l'analyse. Soit l'outil remplace ces appels par des modèles simplifiés. Par conséquent soit l'analyse est très coûteuse ou soit elle requiert une quantité importante de développement pour obtenir un modèle complet.

2.4 Analyse statique de binaire

Sur le principe, l'analyse statique de binaire ne diffère pas de celle basée sur un code source. Le programme n'est pas exécuté et l'analyse requiert toujours un CFG. La reconstruction d'un CFG à partir d'un binaire est reconnu comme étant un problème difficile [3, 7]. En pratique, le CFG reconstruit n'est souvent qu'une approximation du véritable CFG du programme. Les causes de cette imprécision sont la présence de saut dynamique (`" jmp eax "`), de code auto-modifiant ou encore de la possible modification de l'adresse de retour d'une fonction au cours d'une exécution [7].

Ces difficultés disparaissent, du moins en partie, au moment où le programme est exécuté concrètement. En effet si l'analyse est dynamique, toutes les informations présentes au sein du contexte d'exécution sont à la disposition de l'outil pour la trace courante. Ceci permet un ensemble de simplifications que nous détaillons dans le prochain chapitre.

CHAPITRE 3

Exécution symbolique dynamique (DSE)

L'exécution symbolique dynamique (DSE) connue aussi sous le nom d'exécution concolique, pour contraction de concret et de symbolique, est plus récente que son homologue statique. Au début des années 2000, l'arrivée de DART [27] marque le début du développement de cette technique. Celui-ci sera suivi par de nombreux autres outils tels que CUTE [45], CREST, EGT, EXE, KLEE [14], S2E [21], SAGE [28] *etc*.

Actuellement, la plupart des outils exploitant l'exécution symbolique dans d'autres buts que la vérification logicielle, comme par exemple la génération automatique de tests ou la détection de vulnérabilités, se basent en réalité sur la DSE. Cette préférence pour le contexte dynamique s'explique par les solutions, ou plutôt compromis, qu'apporte la DSE face aux challenges rencontrés par l'exécution symbolique classique (Section 1.5). Les avantages et les limites de la DSE seront détaillés après la présentation des différents approches de DSE.

3.1 Les différentes approches

Deux approches tendent à se distinguer aux seins des différents projets dans la manière d'aborder l'implémentation de la DSE : L'*Execution-Based Symbolic Execution* et la *Forked-Based Symbolic Execution*. La distinction entre les deux apparaît au niveau de la granularité choisie pour traverser l'arbre d'exécution : l'une considère tout un chemin tandis que l'autre travaille au niveau des *basic blocks* [1] (BBL). Chacune d'elle possède plusieurs dénominations à travers les différents articles traitant d'exécution symbolique. Nous utilisons, ici, celles données par Chen et *al.* [18]. Les autres dénominations sont rappelées

1. Un *basic block* est une suite d'instruction possédant un unique point d'entrée et un unique point de sortie. De ce fait, si la première instruction du bloc est exécutée, les autres sont exécutées obligatoirement en suivant.

par soucis de cohérence avec les différents articles que le lecteur a pu ou sera mené à lire.

Execution-Based Symbolic Execution

La première est connue sous le nom d'*Execution-Based Symbolic Execution* [18] (EBSE), de *Traced-Based Execution* [41] ou encore d'*Offline Symbolic execution* [17]. Cette approche correspond à celle adopter par DART [27] et l'ensemble des projets influencés par celui-ci : SAGE [28], Bitfuzz [36], Fuzzgrind [16] *etc*.

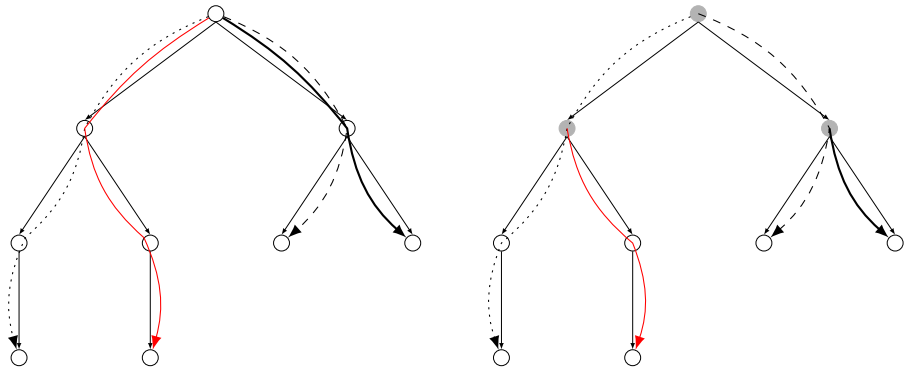
L'exécution symbolique est réalisée en parallèle d'une exécution concrète, il faut donc fournir de véritables arguments au programme qui serviront de *graine* à l'exécution concolique. Celle-ci suit l'exécution concrète, instructions par instructions accumulant au fur et à mesure les contraintes nécessaires à l'élaboration du *path condition*. L'*état* symbolique suit simplement l'exécution concrète. C'est-à-dire qu'en cas de branchement, le *path condition* est mise à jour avec la condition correspondant au saut concret. A noter qu'aucun *fork* n'est réalisé, seul un *état* existe à tout moment. Une fois la trace concrète achevée, l'ensemble des contraintes récupérées permet de construire une nouvelle entrée du programme. En faisant la négation d'une des contraintes, il est possible de construire le *path condition* d'un nouveau chemin. Puis à l'aide de la solution fournie par le SMT solver, une entrée exerçant ce chemin est identifiée. Le programme est réexécuté depuis le début avec cette nouvelle entrée afin d'agréger de nouvelles contraintes, celles-ci serviront à construire une autre entrée et ainsi de suite. Les chemins sont traités les uns après les autres, comme le montre la figure 3.1(a), jusqu'à reconstruire l'arbre d'exécution en entier ou jusqu'à l'épuisement du temps imparti. La stratégie de parcours de l'arbre d'exécution choisit le prochain candidat.

Fork-based symbolic execution

La seconde approche possède les dénominations suivantes : *Fork-based symbolic execution* [18] (FBSE) ou *Online symbolic execution* [17].

Klee et les autres projets basés à partir de lui sont de bons représentants de cette catégorie : Klee-mc [42], S2E [21] et Cloud9 [13, 33] par exemple. Cette approche entrelace l'exécution symbolique et concrète. C'est une version dynamique de l'exécution symbolique « classique » décrite plus haut (section1.2). Une exécution symbolique débute à partir d'un *état* où l'`ip` est l'*entry point* du programme, le `vs` affecte des variables symboliques aux entrées de l'utilisateur et le `pc` vaut *true*. La seule différence réside dans la vérification effectuée avant l'exécution d'une instruction. Si toutes les données impliquées sont concrètes, l'instruction est exécutée concrètement sinon l'exécution est symbolique. Pour ce faire, les *états* doivent aussi stocker le contexte concret. Il est possible de les comparer à des processus agrémenter des éléments de l'exécution symbolique, c'est comme ça que EXE [15] les conçoit d'ailleurs. Les relations de parenté entre ces *états* font apparaître l'arbre d'exécution du programme.

A chaque instruction, une stratégie de parcours choisit depuis quel *état* l'exécution doit repartir. L'exécution n'est liée à aucun chemin par défaut, la stratégie peut très bien choisir un *état* à l'autre bout de l'arbre d'exécution sans



(a) *Execution-based symbolic execution* : parcours de l'arbre d'exécution.

(b) *Fork-based symbolic execution* - Les nœuds gris sont à l'origine des *forks*.

FIGURE 3.1: Parcours d'un arbre d'exécution selon les deux approches de l'exécution dynamique symbolique

pour autant que le dernier *état* exécuté est atteint la fin du programme ou un bug.

Comparaison

Chacune des deux approches possèdent des inconvénients et des avantages. L'approche FBSE implique une plus grosse consommation de la mémoire, car plusieurs *états* peuvent coexister en attendant d'être traités. Les nœuds gris au sein de la figure 3.1(b) correspondent à la rencontre d'un branchement et donc à la création d'un nouvel *état*. Dans cet exemple, il y a trois *forks*, il peut donc y avoir jusqu'à quatre *états* coexistants en même temps. Le *path explosion problem* devient donc un problème à la fois dans le temps et dans l'espace. De plus, l'outil doit être capable de passer d'un contexte concret à un autre puis de relancer l'exécution. Une implémentation optimale d'un *état* n'est donc pas triviale [14, 21].

A l'opposé, l'approche EBSE ne possède à tout moment qu'un seul *état* symbolique qui suit en parallèle le processus du programme analysé. L'impact au niveau de la mémoire est significativement moindre et l'implémentation plus simple [19]. Le contexte concret est directement accessible grâce au processus instrumenté.

L'EBSE possède néanmoins un désavantage. Elle recommence l'exécution depuis le début à chaque fois et donc les préfixes communs à plusieurs chemins sont réexécutés inutilement de façon concrète et symbolique. La figure 3.1(a) met en évidence ce comportement. Les 4 chemins partent tous de la racine de l'arbre et les arêtes partant de celle-ci sont traversées deux fois chacune.

L'autre différence entre les deux approches réside dans la granularité du parcours de l'arbre d'exécution. Une granularité plus fine laisse place à une plus grande flexibilité. Par exemple, l'implémentation FBSE à la possibilité d'interrompre le parcours d'un chemin si celui ci est jugé inintéressant alors que dans une approche EBSE une fois entamée l'exécution se poursuit jusqu'à la fin du chemin en question.

La frontière est poreuse entre les deux approches et des implémentations « hybrides » existent. Mayhem [17] propose de combiner les deux approches afin de réduire l'impacte sur la mémoire (EBSE) sans pour autant devoir ré-exécuter le programme depuis le début (FBSE). Mayhem commence par une approche FBSE. Lorsqu'un certain seuil d'utilisation de la mémoire est atteint, il choisit un *état*, le suspend, puis le conserve dans un point de sauvegarde (*checkpoint*) au sein d'une base de données. L'outil passe alors en mode EBSE est ne « fork » plus aucun nouvel *état*. Les *états* actifs restant sont alors exécutés à la suite jusqu'à épuisement. Quand cela se produit, Mayhem sélectionne une *état* suspendu dans la base de données puis le restaure. Un *checkpoint* contient toutes les informations du contexte symbolique ainsi que les entrées concrètes construites à l'aide du *path condition*. Ces entrées permettront d'exécuter concrètement le programme jusqu'au point sauvegarder puis de relancer une EBSE.

3.2 Avantages et limites

A la différence du SSE, la DSE opère sur une trace du programme. Le programme, ou une partie, s'exécute donc concrètement. A l'aide d'une instrumentation, la DSE accède à l'état concret de l'exécution en cours : les valeurs de la mémoire et des registres ou encore le résultat d'un saut conditionnel. Toutes ces informations simplifient l'exécution symbolique.

Dans une trace d'exécution les boucles sont déroulées, c'est à dire que le corps de la boucle apparaît plusieurs fois de suite au sein de la trace. La DSE n'a pas à décider arbitrairement quand sortir d'une boucle, la trace lui indique cela. Dans le cas suivant `" if (x == hash(y)) "`, la SSE est bloquée car le SMT solver est incapable de dire si cette condition est satisfaisable ou non. Par contre avec la DSE, pour une valeur fixée de `y`, toute la partie de la formule correspondant à `" hash(y) "` peut être remplacée par le résultat concret de la fonction de hachage (concrétisation). La formule est donc simplifiée ($x =$ Constante) et elle redevient à la portée du SMT solver. Cette même solution est applicable lorsque la manipulation de pointeur symbolique devient trop complexe ou si l'outil ne souhaite pas gérer tous les valeurs possibles du pointeur.

A noter que la concrétisation est un compromis. Certes, cette simplification est peu couteuse et permet de poursuivre l'exécution symbolique en évitant certains problèmes difficiles. Cependant, elle entraine une sous-approximation en ce qui concerne le parcours de l'arbre d'exécution. L'outil rate certaines branches de l'arbre car une seule valeur est récupérée du contexte concret au lieu de l'ensemble des valeurs possibles représentées par l'expression symbolique. Dans le pire des cas, la concrétisation peut même entrainer des divergences au sein de l'exécution symbolique.

Une divergence correspond à une séparation du flot de contrôle entre la trace symbolique et celle concrète, typiquement les deux traces n'empruntent pas la même branche d'un saut conditionnel alors qu'elles le devraient [19].

Cela correspond à une erreur au sein du *path condition* généré pour ce chemin. L'exemple suivant (Listing 3.1) illustre comment la concrétisation appliquée à l'arithmétique des pointeurs peut entrainer des divergences. Nous réalisons une exécution concolique de la fonction `divergence` où les variables x

Listing 3.1: Fonction divergence

```
1 void divergence(int x, int y) {
2    int s[4];
3    s[0] = x;
4    s[1] = 0;
5    s[2] = 1;
6    s[3] = 2;
7
8    if (s[x] == s[y] + 2)
9       abort();
10 }
```

et y sont définis comme étant symbolique. Prenons $x = 0$ et $y = 2$ comme entrées. Une fois arriver à l'instruction ligne 8, la condition dans le domaine symbolique est égale à $s[x] == s[y]+2$. x et y étant symbolique, les deux pointeurs $\&s[x]$ et $\&s[y]$ peuvent prendre une multitude de valeurs. Au lieu de toutes les considérer, la valeur de chaque pointeur est concrétisée. La condition devient donc égale à $s[0] == s[2] + 2 \Leftrightarrow x == 3$. Dans le contexte concret actuel, cette condition est évidemment fausse ($0 \neq 3$) par conséquent la contrainte récupérée est $\phi = (x \neq 3)$. Pour atteindre l'instruction ligne 9 où l'erreur se produit, nous générons une nouvelle entrée à partir de la négation de ϕ. Ce qui nous donne $\neg\phi = (x = 3)$. L'exécution concrète avec les entrées $x = 3$ et $y = 2$ devrait donc atteindre l'instruction ligne 9. Or, même ici, la condition ligne 8 n'est pas satisfaite : $s[3] == s[2] + 2 \Leftrightarrow 2 == 3$. Il y a donc ici une divergence, le flux d'exécution concret ne correspond pas au flux d'exécution prévu par l'exécution symbolique. Les entrées construites à partir de l'exécution symbolique précédente ne sont pas correctes et ne nous permettent pas d'atteindre l'instruction ligne 9. La contrainte récupérée n'est donc pas la bonne.

La gestion de l'arithmétique des pointeurs reste un problème ouvert. Le projet Mayhem [17] propose d'exécuter une *Value Set Analysis (VSA)* en parallèle de l'exécution symbolique afin de calculer des ensembles sur-approchés des valeurs des pointeurs. D'après le même article, cet ensemble reste suffisamment petit en pratique pour considérer une analyse au cas par cas. Cette solution rend donc l'exécution symbolique plus complète en pratique même si théoriquement elle est toujours sujette à une explosion combinatoire.

L'interaction avec l'environnement extérieur est aussi problème pour l'exécution symbolique où la DSE est capable d'apporter des solutions. Dans un contexte dynamique, le résultat des interactions avec l'environnement est connu pour des valeurs concrètes. Dans le cas où des variables symboliques sont impliquées, la concrétisation est donc, ici aussi, une solution envisageable. Selon l'instrumentation utilisée et le langage ciblé, l'ensemble de l'environnement peut être exécuté symboliquement : de l'application jusqu'au noyau en passant par les signaux et les bibliothèques partagées [21]. Ce type d'exécution symbolique est très intéressante car elle est complète dans le sens où toutes les instructions effectuées par le processeur sont prises en compte. Évidemment ce type d'exécution est beaucoup plus couteuse en termes de ressources et aussi plus lente car plus d'instructions sont traitées. Il y a un changement d'échelle notable entre l'analyse d'un unique programme et celui de tout un

ensemble de code formé par le système d'exploitation et le programme cible. La modélisation de l'environnement reste une solution. C'est celle adoptée par les outils tel que Klee[14] ne pouvant pas réaliser ce type d'instrumentation et qui ne souhaitent pas de la sous-approximation induite par la concrétisation.

Une autre différence notable entre SSE et DSE réside dans le nombre de variables symboliques nécessaires au cours de l'exécution. Typiquement, toutes variables issues de l'environnement extérieur a une valeur indéfinie pour une SSE. Elles se voient donc attribuer une valeur symbolique, sinon l'exécution symbolique ne peut pas continuer. Or certaines entrées provenant de l'environnement peuvent ne pas intéresser l'analyste et c'est donc dommageable de perdre en performance pour des informations au final inutiles. C'est le cas par exemple des variables d'environnement ou d'un fichier de configuration lors d'une analyse de sécurité. Les entrées de l'utilisateur ou les données issues d'une *socket* au contraire sont à privilégier. La DSE offre la possibilité de restreindre l'exécution symbolique aux seules données intéressantes pour l'analyste, pour toutes les autres leurs valeurs concrètes sont utilisées.

Même si la DSE ne souffre pas des problèmes du SSE, d'autres challenges apparaissent. Le plus imposant, qui est d'ailleurs abordé par la quasi-totalité des articles [44, 14, 27, 28, 33, 21] parlant d'exécution symbolique dynamique, est le *path explosion problem*. Ce problème se manifeste sous la forme d'un temps calcul extrêmement long ou de l'épuisement de la mémoire dans le cadre d'une FBSE [17].

3.3 Explosion combinatoire : exemples d'optimisations

Cette section répertorie les différentes optimisations ayant pour but de permettre à l'exécution symbolique de traiter des programmes de tailles toujours plus importantes. Nous les présentons dans cet ordre. Nous commençons par les heuristiques améliorant le parcours au sein de l'arbre d'exécutions. Puis les techniques permettant de sélectionner uniquement les instructions nécessaires au bon déroulement de l'exécution symbolique. Nous considérons par la suite les solutions hybrides alliant *fuzzing* [2] et DSE. Nous continuons en abordant des versions de l'exécution symbolique prenant en compte la structure du programme pour éviter d'exécuter plusieurs fois les parties similaires de l'arbre d'exécution. Enfin, nous nous penchons sur les approches liées au calcul parallèle et au architecture distribuée.

Stratégie de parcours de l'arbre

Une première optimisation consiste à adapter le parcours de l'arbre d'exécution à notre analyse. Il a été démontré que les parcours d'arbres triviaux : parcours en profondeur (*Deep first search (DFS)*) et parcours en largeur (*Breadth first search (BFS)*) n'étaient pas adaptés aux problèmes [28, 44]. Pour le premier, il est possible d'arriver jusqu'à l'arrêt du programme ou de trouver un bug mais au final la couverture de code obtenue est peu importante dans le laps de temps imparti [14]. Par ailleurs, sans une limite d'exécution du corps d'une

2. Le *fuzzing* est une technique de test logiciel. Elle consiste à exécuter un programme avec un grand nombre de valeurs choisies de manière aléatoire ou l'aide d'heuristiques.

boucle, ce type de parcours est inefficace pour les boucles où la condition d'arrêt comprend des variables symboliques. Ce parcours tend à dérouler jusqu'à la dernière itération possible la boucle en question, ce qui peut être long voire pire si c'est une boucle infinie [15, 44]. L'exécution symbolique réexécute un grand nombre de fois la même portion de code, ce qui n'est pas optimale en termes de couverture de code. . Quant au second, c'est l'inverse. Ce parcours tend à exercer différentes parties du code mais ne va pas assez profondément pour atteindre un crash ou l'arrêt du programme.

L'autre point négatif de ces deux parcours mis en avant par P. Godefroid [28] est qu'un seul nouveau test est généré pour chaque chemin parcouru. Ce qui est un piètre rendement vu le prix d'une exécution symbolique. C'est la raison pour laquelle, SAGE utilise la stratégie appelée *generational search*. A la fin d'une exécution, chaque contrainte composant le *path condition* et sa négation sont prises en compte avant de générer n nouveaux tests où n est le nombre de nouvelles contraintes découvertes lors de l'exécution. Klee, quant à lui, repose sur une combinaison de deux heuristiques appliquées à tour de rôle. La première heuristique traverse l'arbre binaire des *états* depuis la racine en choisissant une branche aléatoirement. La seconde heuristique tente de choisir *l'état* susceptible de favoriser la couverture de code. Cette heuristique associe un poids à chaque *état* basé sur différents critères et choisis celui avec le meilleur.

Exécution symbolique restreinte ou sélective

Certaines instructions du programme n'agissent pas sur les entrées de l'utilisateur (variables symboliques), il est donc inutile d'essayer de les exécuter symboliquement et de générer des expressions composées seulement de constantes. Mayhem [17] se base sur une analyse de teinte dynamique afin de filtrer ce type d'instruction. Klee réalise la même chose sans l'aide d'analyse supplémentaire car cette information (opérande symbolique ou non) est stockée au sein des *états*.

S2E introduit la notion de *selective symbolic execution* [20]. Celui-ci voit le système entier (application, noyau, bibliothèques partagées *etc.*) comme un seul programme, au sens de suite d'instructions à exécuter, où seul les parties intéressantes, selon l'utilisateur, sont exécutées symboliquement. S2E peut convertir l'argument d'une fonction ou d'un appel système d'un domaine à l'autre (Concret $\leftrightarrow$ Symbolique). Ces conversions suivent un *consistency model* permettant de configurer leur impact sur l'exécution en termes de « cohérence », d'« exhaustivité » et de temps de calcul [21].

De manière général, un modèle relâché est moins coûteux mais complet (il considère tous les chemins possibles). Cependant, il introduit le risque de voir apparaître des faux positifs (chemins infaisables en pratique). A contrario, un modèle strict donc plus proche de la réalité est plus précis mais demande une instrumentation plus importante.

Par exemple, considérons le cas d'un appel système. Si l'exécution symbolique se poursuit au sein du noyau, la précision de l'exécution symbolique sera précise mais cela nécessite d'instrumenter du code supplémentaire propre à l'environnement. A l'inverse si la valeur de retour de l'appel système est remplacée par un nouveau symbole, l'exécution symbolique reste en *user-land* et par conséquent moins de code est instrumenté. Par contre ce nouveau sym-

bole n'est pas contraint et il peut donc prendre des valeurs normalement impossibles dans le cadre d'une exécution concrète (faux positif).

Le choix du modèle dépend du contexte d'utilisation de l'exécution symbolique et des objectifs. La génération automatique de tests, le *reverse engineering* ou la vérification n'ont pas les mêmes besoins en termes d'exhaustivité et de précision.

Fuzzing et exécution symbolique dynamique

L'exécution symbolique sélective tend à limiter le plus possible, les parties du code devant être exécutées symboliquement. Son but à termes est reste d'essayer de parcourir l'ensemble de l'arbre d'exécution.

Dans le cadre de la recherche de vulnérabilités, l'objectif n'est pas d'exercer l'ensemble des chemins possibles d'un programme mais plutôt de trouver les chemins susceptibles de déclencher un bug (*corner case*).

En prenant cela en compte, il devient intéressant de délaisser certains chemins ou sous-arbres au profit d'autres juger plus intéressants (*path pruning*). Pour ce faire des projets [35, 39, 18] allient *fuzzing* et exécution symbolique. L'idée est de profiter des points forts de chaque technique. L'exécution symbolique est capable de parcourir un sous-arbre sans jamais passer deux fois par le même chemin. Par contre, quelque soit la stratégie utilisée, celui-ci se fait de manière locale, car les nouveaux chemins sont créés à partir des contraintes précédemment trouvées. A l'inverse, le *fuzzing* peut exercer plusieurs fois un même chemin dans l'arbre d'exécution mais il a aussi la capacité à exercer des chemins éloignés plus facilement : en faisant varier la taille de l'entrée par exemple. L'autre avantage du *fuzzing* est sa rapidité en comparaison de l'exécution symbolique.

La stratégie de Majumdar et Koushik [35] consiste par commencer à exécuter concrètement le programme avec des entrées aléatoires jusqu'à atteindre un point de convergence. Cela signifie que la méthode aléatoire n'arrive plus à couvrir de nouvelle portion du code et donc ne peut plus accroître le pourcentage de couverture. A ce moment là, l'outil prend un instantanée de l'état concret du programme puis réalise une exécution concolique à partir de ce point. L'exécution concolique s'arrête si celle-ci a trouvé une entrée permettant d'augmenter la couverture de code ou si le temps imparti est épuisé. L'outil continue alors sa phase de *fuzzing* là où elle s'était arrêté avec l'entrée issue de l'exécution concolique comme nouveau point de départ.

L'approche évoquée dans la thèse de B. Pak [39] prend le parti pris inverse. L'analyse débute par une exécution dynamique symbolique (FBSE). Elle s'arrête si le temps imparti est épuisé ou si un seuil sur le nombre d'*état* est atteint. Pour chaque *état*, une entrée est générée. Ces cas de test *partiels* servent de graines pour un *fuzzer*. L'idée est de préserver les parties du cas de test dépendantes du *path condition*. Le reste du fichier peut par contre être modifié sans restriction et il est donc possible de lui appliquer différents types de mutations. *Crashmaker* [18] reprend ce même concept mais celui-ci ne comporte pas de phase de *fuzzing*. Les mutations sont effectuées directement par l'exécuteur symbolique dynamique au moment de la génération d'une nouvelle entrée.

Il existe une autre différence entre les travaux de B. Pak et ceux de Chen et *al.* (*Crashmaker*). *Crashmaker* se contente de laisser intacte les octets dont la

valeur est issue de la réponse du SMT solver. Cela signifie qu'il garde un représentant dans la classe d'entrées représentée par le *path condition* et qu'il *fuzz* juste le reste des octets n'intervenant pas dans la formule. La solution de B. Pak *fuzz* l'intégralité de l'entrée en faisant en sorte que celle-ci reste dans l'ensemble des solutions du *path condition* produit par l'exécuteur symbolique. Utiliser le SMT solver pour énumérer le modèle est inefficace, faire une requête par élément est beaucoup trop coûteux. Pak propose donc de linéariser le *path condition*. C'est-à-dire de remplacer les égalités par des inégalités afin d'obtenir le modèle comme une union de plusieurs intervalles. Par exemple l'ensemble $S = \{1, 2, 3, 7, 8\}$ peut être vu comme l'union des deux intervalles $S = [1, 2] \cup [7, 8]$. Il est ainsi plus rapide de s'assurer qu'un octet respecte certaines bornes plutôt que de vérifier son appartenance à un ensemble. Cette technique a le défaut majeur de ne pouvoir traiter que des formules linéaires. Ici linéaire signifie que la formule ne peut contenir que des constantes et des variables du premier ordre avec des coefficients constants. Donc $4 \times \alpha + 12 = 56$ est linéaire mais pas $\alpha \times \beta < 4$. Selon la manière adoptée sur ou sous-approximation, une formule non linéaire peut soit entraîner la création de cas de tests ne respectant pas le *path condition* ou soit perdre la capacité de générer certaines entrées.

State merging et *Summary*

Les optimisations suivantes tentent de prendre en compte la structure du programme afin de limiter le nombre de chemins à parcourir. En analysant le CFG d'un programme, il est facile de voir que certaines instructions séparent le flux d'instruction en plusieurs branches (plusieurs arêtes sortantes), comme les sauts conditionnelles, alors que d'autres sont des points de convergences pour plusieurs flux d'exécution (plusieurs arêtes entrantes).

La figure 1.2 comporte deux points de séparation : le nœud `" if (x < 0) "` et le nœud `" if (x > 15) "` ainsi que deux points de jonction : le nœud `" if (x > 15) "` et le nœud `" <return> "`. Supposons que l'exécution symbolique de la fonction *scoring* dispose de deux *états* dont l'`ip` vaut 7, c'est-à-dire l'instruction `" if (x > 15) "` l'un possède un `pc` valant $\alpha < 0$ et l'autre un `pc` valant $\alpha \geq 0$. Comme ces deux *états* sont à la même instruction, ils vont exécuter symboliquement le même code chacun de leur côté et récupérer les mêmes contraintes. Il serait plus intéressant de fusionner (*merge*) ces deux *états* en un seul puis de continuer l'exécution. Nous évitons ainsi toute redondance. Ce nouvel *état* serait donc :

`ip:` 7
`vs:` $(y \rightarrow \text{ITE}(\alpha < 0, -\alpha + 1, 0)$
`pc:` $(\alpha < 0 \vee \alpha \geq 0) = true$

Le `pc` du nouvel état est à la disjonction des deux états fusionnés.

De manière informel, pour arriver à cet endroit du code, nous pouvons passer par deux chemins : celui où α est strictement inférieur à 0 **ou** celui où α est supérieur ou égal à 0. Le `vs` prend en compte les deux valeurs possibles de y, une fois la ligne 7 atteinte. L'expression ITE ici signifie que y vaut $\alpha + 1$ si nous avons suivis le chemin induit par $\alpha < 0$ ou y vaut 0 si nous avons suivis l'autre chemin. Cette fusion est équivalente au *static state merging* [33] qu'utilise les *verification condition generator* qui ont déjà été présentés dans la section sur le SSE (2.2).

Cette technique permet d'éviter de traiter plusieurs fois une même partie de code au sein de différents chemins. Cela implique aussi une réduction du nombre d'*états* à stocker et à traiter. Malheureusement, cette optimisation n'est pas exempte de défauts. Les *états* issues d'une fusion possède un *path condition* plus complexe. Les limites du SMT solver sont donc plus rapidement atteinte ce qui se caractérise par un temps de réponse long ou l'incapacité à donner une réponse.

Les projets Cloud9 [33] et Mergepoint [3] incorpore cette optimisation appelée *state mergin* par Kuznetsov et *al.* [33]. Le *state mergin* qui peut être décrite comme un mélange entre des techniques propres à l'exécution dynamique et statique. La différence entre les deux approches résident dans la façon de détecter les *états* susceptibles d'être fusionnés. Cloud9 utilise une métrique afin de décider quand une fusion est favorable. Cela signifie que le compromis entre la complexité des formules et la réduction du nombre d'*état* est jugé favorable. Quand c'est le cas, Cloud9 favorise certains *états* afin qu'ils arrivent plus vite à la même instruction et puissent donc être fusionnés.

Mergepoint applique une technique similaire, nommée *Veritesting*, de manière systématique dès qu'un branchement est rencontré. Mergepoint reconstruit un CFG partiel partant de l'*état* actuel et s'arrêtant dès qu'une situation trop compliquée à gérer dans un contexte statique est rencontrée. Un saut indirect où l'adresse ne peut être connu statiquement ou encore un appel système (Mergepoint étant basé sur Pin [30], il n'est donc pas capable d'instrumenter le noyau). Ces instructions à la frontière du CFG forment des points de transition. Ensuite Mergepoint applique une exécution symbolique statique sur ce CFG. Pour terminer, un nouvel *état* est créé pour chaque point de transition, qui sont autant de nouveaux points de départs pour l'exécution symbolique dynamique.

Le *state merging* réduit la quantité de travail de l'exécuteur symbolique en supprimant les différentes redondances existantes entre les *états*. La technique des *summaries* permet aussi d'éviter des redondances mais cette fois ci, à l'intérieur même des chemins. Certains bloc d'instruction apparaissent plusieurs fois au sein d'une même trace, cela est dû aux boucles et aux appels de fonctions.

Les *summaries* [2] résument sous la forme d'une formule les effets d'un bloc d'instructions. L'idée étant d'utiliser ce *summary* en lieu et place des prochains occurrences de ce même bloc au sein de l'exécution symbolique. Une *function summary* ϕ_f est définie comme la disjonction de tous les formules ϕ_p qui correspondent aux formules issues des chemins parcourant la fonction.

$$\phi_f = \bigvee_p \phi_p$$
$$\phi_p = \text{pre}_p \wedge \text{post}_p$$

Pour un chemin p, pre_p est la conjonction des contraintes portant sur les entrées. post_p est la conjonction des contraintes portant sur les sorties [2].

Par l'exemple, la *function summary* de la fonction *scoring* (listing 1.1) est

$$\phi_{scoring} = \underbrace{(\overbrace{X < 0}^{\text{pre}} \wedge \overbrace{Y = -X + 1}^{\text{post}})}_{\phi_{p_1}} \vee \underbrace{(\overbrace{X > 15}^{\text{pre}} \wedge \overbrace{Y = 2}^{\text{post}})}_{\phi_{p_2}} \vee \underbrace{(\overbrace{X \geq 0}^{\text{pre}} \wedge \overbrace{Y = 0}^{\text{post}})}_{\phi_{p_3}}$$

Les formules ϕ_{p_i} correspondent aux nœuds bleus de l'arbre d'exécution (figure 1.1) qui sont les points de sorties de la fonction *scoring*. Les variables X et Y servent ici de remplaçants provisoires en attendant les arguments et la variable contenant le résultat de la fonction. Ce ne sont pas des variables symboliques correspondant aux entrées de l'utilisateur. Le *summary* doit, en effet, pouvoir être utilisé pour n'importe quel appel à la fonction *scoring*.

Par exemple, considérons l'appel suivant `" int tmp = scoring(z) "` où z est contient l'expression symbolique $\alpha \times \beta - 6$. A l'aide de la *function summary* précalculée, la valeur symbolique de *tmp* est donnée directement par la formule :

$$\underbrace{\texttt{tmp} = Y}_{\text{résultat}} \wedge \underbrace{X = \alpha \times \beta - 6}_{\text{argument}} \wedge \phi_{scoring}$$

Cette formule peut être réecrite à l'aide d'expressions ITE puis associée à `tmp`.

$$\texttt{tmp} \rightarrow \text{ITE}(X < 0, -X + 1, \text{ITE}(-X > 15, 2, 0)) \wedge X = \alpha \times \beta - 6$$

De la même manière que le *state merging*, les *summaries* complexifient les formules envoyés aux SMT solver. Les *summaries* insèrent des disjonctions au sein des formules comme le montre l'exemple précédent. Le compromis se situe, ici aussi, entre la complexité de la formule du *summary* et le nombre de répétitions évitées par l'exécution symbolique. Pour une fonction appelée à de multiples reprises au cours d'une exécution, ajouter un *summary* au *path condition* est plus efficace que d'exécuter symboliquement plusieurs fois le même appel à la fonction. Insérer ce même *summary* comprenant tous les chemins possibles au sein de la fonction pour un seul appel est à l'inverse moins performant. La complexité ajoutée au *path condition* est alors inutile. Cette idée est présentée par Kuznetsov et *al.* dans l'article traitant de Cloud9 [33]. L'article en question fait remarquer que le surcout entrainer par la création automatique des *summaries* peut aussi réduire les performances dans certains cas. Cela est dû notamment à l'instrumentation nécessaire pour couvrir le code de la fonction.

Throw hardware to your problem

Malgré l'ensemble des optimisations vu jusqu'à présent, l'exécution symbolique demande toute de même une quantité importante de ressources. De plus ces optimisations n'assurent pas pour autant l'exhaustivité du parcours de l'arbre d'exécution. C'est pour cela que les exécuteurs symboliques possèdent un temps limité pour réaliser leur analyse. Toutes les optimisations énoncés précédemment sont utiles pour utiliser au mieux ce temps imparti. L'autre approche basée sur la parallélisation permet de travailler sur plusieurs chemins (ou sous-arbres) en même temps. SAGE, S2E et Cloud9 sont trois projets mettant en avant cette approche.

Chacun des différents composants de SAGE (AppVerifier, Nirvana ...) peuvent être exécutés en parallèles sur une machine multi-cœur [10]. S2E permet lui aussi de bénéficier de l'avantage d'une architecture multi-cœur, en créant une ou plusieurs autres instances de lui-même et en partageant l'arbre d'exécution entre toute ces instances. Cloud9 lui pousse l'idée plus loin. Le projet repose sur un *load balancer* répartissant dynamiquement la charge de

travail (à savoir des sous-arbres de l'arbre d'exécution) à des *workers* qui sont des exécuteurs symboliques dynamiques (des instances de Klee modifié).

D'autres projets comme SAGE et Mergepoint ont eu aussi l'idée d'intégrer des exécuteurs symboliques au sein d'une architecture distribuée. A la différence de Cloud9 où le parallélisme sert à optimiser une exécution symbolique, ces projets ont construis des architectures distribuées afin de tester en masse différents programmes. L'idée étant d'avoir un *cluster* ou un *cloud* tournant 24h/24, 7j/7 testant à la chaîne plusieurs logiciels.

SAGE fait, aujourd'hui, parti intégrante du processus de tests des logiciels Microsoft au même titre que leur cluster dédié au *fuzzing* [28, 10]. Quant à Mergepoint, il a été utilisé pour tester l'ensemble de la suite logiciel *coreutils* sur la distribution Debian. Mergepoint a reporté à termes plus de 1500 crashs.

3.4 Composition d'un exécuteur symbolique de binaire dynamique

Dans cette section, nous présentons les différents composants logiques et logiciels présents aux sein d'un exécuteur symbolique. Nous nous concentrons en particulier sur le cas de l'exécution symbolique de code assembleur. Nous commençons par voir les différentes solutions rencontrées afin de récupérer une trace d'exécution. Puis comment celle-ci peut être traitée, traduite et exécutée symboliquement. Pour finir, nous réalisons un point, sur la brique vitale que représente le SMT solver au sein d'un exécuteur symbolique.

Dynamic Binary Translator (DBT)

Le DBT [42] ou *Dynamic Binary Translator* est investi de deux missions : 1) il récupère dynamiquement la trace d'exécution concrète et 2) il traduit le code assembleur vers la représentation intermédiaire utilisée par le moteur d'exécution symbolique.

La première phase est réalisée au travers d'outil capable d'instrumenter dynamiquement un binaire tel que Pin [30], Valgrind [38] ou QEMU [9]. La récupération de la trace peut s'effectuer de deux façons. Soit le DBT récupère d'une traite la trace et la stocke sur le disque sous la forme d'un fichier qui est lu ensuite par le moteur d'exécution symbolique. Soit chaque instruction est récupérée, traduite puis envoyée au moteur à la volée (*just-in-time*) tout au long de l'exécution.

La seconde approche est bien évidemment la plus légère et rapide, elle ne nécessite pas d'accès au disque et le moteur récupère les informations nécessaires en fonction du contexte concret actuel. Au contraire, la première approche nécessite de stocker toutes les informations potentiellement intéressantes pour le moteur : instruction, opérande, état des registres et la mémoire. Ce qui entraine un fichier volumineux même pour une seule trace [10].

La première approche possède, tout de même, deux avantages. Le fait de figer la trace sous une forme sérialisée évite certains problèmes liés au non-déterminisme pouvant exister au sein d'un programme. Dans le cadre d'un programme avec plusieurs threads, des événements imprévus (une *race condition* par exemple) peuvent modifier le flot d'exécution de la trace pour une même entrée. Ce problème peut se matérialiser sous la forme de divergence

lors d'une EBSE par exemple. L'exécution concrète ne suit pas le chemin correspondant au *path condition* calculé à partir d'une trace précédente. Certaines entrées générées par l'outil peuvent être prises alors à tort pour des faux positifs.

Considérons le scénario suivant : l'outil détecte un bug au sein d'un programme multi-threadé et créé une entrée correspondante. Afin de s'en assurer, nous exécutons le programme concrètement avec cette même entrée cependant le bug n'apparaît pas et le programme termine normalement. Cela semble être un faux positif. Or l'outil a bien trouvé un véritable bug. La présence d'un événement non déterministe modifie le déroulement du programme le rendant variable pour une même entrée. Ceci rend difficile l'analyse de ce bug à l'aide d'une simple exécution concrète. Sans une étude plus approfondie, il est difficile de différencier ce cas de figure d'un véritable faux positif. Le second avantage est d'ordre architectural. Le moteur d'exécution d'exécution n'est pas lié à la DBT car celui-ci travaille à partir d'un fichier. La plateforme, le système d'opération et le framework n'influencent en rien la capacité du moteur à exécuter symboliquement la trace [10]. Seul l'outil générant la trace se doit d'être adapté à la plateforme où s'exécute le binaire cible.

La phase de traduction est spécifique à chaque projet. Certaines traductions [42, 21] passe par plusieurs étapes avant d'arriver à la représentation utilisée par le moteur d'exécution symbolique. Par exemple, S2E repose sur QEMU. Celui-ci traduit l'assembleur du programme vers sa représentation puis une deuxième traduction vers le bytecode LLVM est effectuée. Cela permet ainsi à une version de Klee modifiée de réaliser l'exécution symbolique.

Représentation Intermédiaire et moteur d'exécution

La plupart des exécuteurs symboliques de binaire n'expriment pas directement de façon symbolique le code assembleur. La principale raison est la quantité d'instructions existantes en particulier pour les assembleurs CISC [3] (x86, x86_64). Les représentations intermédiaires (IR) ou langages intermédiaires (IL) possèdent un jeu d'instruction plus restreint (SimplIL [44], BIL [12], LLVM bytecode [34]). En effet, travailler directement sur le langage assembleur implique d'écrire une version symbolique de chaque instruction. C'est une tâche fastidieuse et propice aux erreurs. De plus, cela limite l'exécuteur symbolique a un seul langage assembleur. Un nouvel outil doit donc être développer pour chaque autre architecture. L'avantage directe d'utiliser un IR est donc de découpler le moteur d'exécution symbolique du langage du programme cible. Ce principe déjà bien connu vient du monde des compilateurs. Ceux-ci se décomposent la plus part du temps en 3 parties :

- la *front-end* : traduit le langage source vers l'IR. Dans le cas de la compilation le langage source est souvent de haut niveau.
- l'IR : le compilateur réalise des optimisations et autres analyses statiques sur celui-ci.
- la *back-end* : traduit l'IR vers le langage de destination, le plus souvent de l'assembleur.

Nous retrouvons, ici, cette décomposition en trois blocs. Le DBT est une *front-end* et l'interface avec le SMT solver (traduction IR vers des formules)

3. Complex instruction set computing

correspond à la *back-end*. Un autre avantage évident d'une telle décomposition est que tous les *front-ends* et *back-ends* bénéficient des optimisations et des améliorations du moteur d'exécution.

La création d'un IR efficace n'est pas triviale à l'inverse de ce que laisse penser Schwartz et *al.* [44]. Sa conception, son implémentation puis les simplifications, optimisations ou autres extensions représentent un travail important. C'est pourquoi il représente parfois un projet à part entière : Bitblaze (Vine) [47] et BAP [12].

En pratique, le plus souvent l'exécuteur symbolique se contente de réutiliser l'IR de la *front-end* : VEX pour Fuzzgrind [16], Vine pour FuzzBall [36] et Bitscope [11], BIL pour Mayhem [17] et Mergepoint [3] ou encore le microcode de QEMU pour S2E [21].

Au final, l'IR, le moteur d'exécution et la *front-end* sont tout de même liés. Le seul projet sortant du lot et qui a pu être utiliser pour traiter de l'assembleur ou un langage plus haut niveau tel que le C est Klee. Celui-ci fonctionne réellement comme un interpréteur d'une version étendue symboliquement du bytecode LLVM. Initialement, le compilateur clang [4] était la seul solution afin d'obtenir le bytecode d'un programme cible écrit en C. Cependant S2E s'est basée sur Klee afin de pouvoir analyser dynamiquement des binaires écris dans différents assembleurs : x86, x86_64 et ARM.

SMT solver

La capacité de résoudre les formules produites par l'exécuteur symbolique influent fortement sur ces résultats. En effet, tout *path condition* ne pouvant pas être traité impacte directement l'exhaustivité du parcours de l'arbre d'exécution, laissant des parties du programme non analysées. Par ailleurs, les résultats attendus par l'utilisateur de la part de l'exécuteur symbolique comme par exemple l'entrée menant à un bug, proviennent des solutions de ces formules. Le rôle du SMT solver est donc très important.

Néanmoins, il est nécessaire de distinguer l'exécution symbolique du problème SMT. Le problème de Satisfaisabilité Modulo Théorie (SMT) est difficile (NP-complet dans le meilleur des cas [4, 37]). C'est la raison pour laquelle la résolution des formules issues de l'exécution symbolique sont déléguées à un outil tiers (*off-the-shelf*) spécialisé appelé *SMT solver* ou *theorem prover*.

Malgré la difficulté du problème auquel s'attaque les SMT solvers, la recherche à ce propos semble dynamique, en témoigne le nombre d'outils existant : Boolector, STP, Yices, Z3 *etc.* Les principaux axes de recherche semblent être la gestion de nouvelles théories (nombres flottants [10]) et l'amélioration des performances des outils actuelles (complexité des formules). Il existe une compétition se tenant chaque année : SMT-COMP [8]. Celle-ci compare les performances de différents SMT solvers selon plusieurs catégories en fonction de la théorie et des logiques utilisées.

Le problème SMT est un sujet de recherche à part entière et l'utilisation des SMT solvers ne se cantonnent pas à l'exécution symbolique. De ce fait, le cœur des travaux traitant de l'exécution symbolique s'intéresse plus aux problématiques évoquées dans les chapitres 2 et 3 plutôt que de se soucier de la façon dont sont résolues les formules. Il faut voir le SMT solver comme

4. `http://clang.llvm.org/`

une dépendance pour l'exécuteur symbolique. Toute avancée dans le domaine entrainera un gain certain pour l'exécution symbolique.

3.5 Conclusion

L'exécution symbolique se décline sous différentes déclinaisons : statique, dynamique, FBSE, EBSE ou comme une combinaison de ces approches. Historiquement, l'approche statique est liée au contexte statique tandis que l'approche dynamique est favorisée par les outils cherchant à faire des tests logiciels ou de l'analyse de *malware*. Il est difficile aujourd'hui d'imaginer une exécution symbolique universelle s'adaptant à tous les cas de figure. Cependant les travaux de Godefroid [2], de Kuznetsov [33] et de Avgerinos [3] font apparaître une tendance dans l'évolution de l'exécution symbolique. Le contexte dynamique est sans nulle doute celui privilégié [26]. Néanmoins pour des questions de performance et de passage à l'échelle, des éléments typiques de la SSE y sont intégrés. La structure du programme est pris en compte (fonctions et CFG) par exemple.

En définitive, l'exécution symbolique est une technique d'analyse qui doit être adaptée au but que l'on souhaite atteindre. Si nous nous limitons à la recherche de vulnérabilités sur des binaires, la présence de sauts dynamiques ou de code auto-modifiant nous oblige à nous tourner vers la DSE.

Deuxième partie

Couplage et boucle de rétroaction entre les deux techniques

CHAPITRE 4

Couplage exécution symbolique et fuzzing : démarche

Dans ce chapitre, nous abordons la réalisation du système alliant un exécuteur symbolique (S2E) et un *fuzzer* (AFL) nommé par la suite Fuzzs2e. Nous commençons par présenter les travaux qui ont inspiré notre système avant de présenter les objectifs que nous nous étions fixés lors de la réalisation de Fuzzs2e. Nous continuons par l'explication du fonctionnement global de Fuzzs2e, en mettant en avant les quelques différences par rapport à l'approche proposée par B. Pak [39]. Puis nous présentons chaque outil, leur place au sein de notre système et expliquons pourquoi notre choix s'est arrêté sur eux. Pour finir, nous détaillons les principales modifications apportées aux deux logiciels et choix d'implémentations que nous avons effectués.

4.1 Inspiration et concept initial

Les travaux de Majumdar et Koushik [35], puis de Pak [39], et enfin ceux de Chen et al. [18] mettent en avant l'utilisation combinée des deux techniques d'analyses dynamiques que sont l'exécution symbolique et le *fuzzing*. L'aspect complémentaire des ces deux techniques apparaît clairement lors de la comparaison de leurs points forts et points faibles.

L'exécution symbolique est lente à cause de l'instrumentation et de la traduction qu'elle requiert. A l'inverse le *fuzzing* ne nécessite, quant à lui, peu voire pas d'instrumentation. Ce faible surcout lui permet de tester le programme cible très rapidement. Du point de vue des performances, ces techniques sont aux antipodes.

Cette différence s'explique aussi par la manière dont le problème de la génération de tests est abordé. Bien que les deux techniques utilisent des traces concrètes du programme, la façon dont l'ensemble des entrées est parcourue, est très différente. Un *fuzzer* « classique » (absence de grammaire) travaillant

sur des données binaires est par défaut totalement indifférent à la structure de l'entrée qu'il génère ou mute. Il n'a pas la vision de l'arbre d'exécution que possède l'exécuteur symbolique et comme il repose plus ou moins sur la chance pour produire ces entrées, rien n'assure celles-ci de ne pas exercer encore et toujours les mêmes parties du programme. L'exécuteur symbolique infère la structure (ou une partie) de l'entrée à l'aide des informations qu'il extrait du flot de contrôle et du flot de donnée. Comme décrit précédemment (1.2), celui-ci découvre de nouveaux chemins au sein du programme de proche en proche, créant de nouveaux *path conditions* à partir des contraintes déjà connues. Bien que précis, ce processus est long et peut être inefficace si la stratégie de parcours adoptée n'est pas adapté au programme.

L'exécuteur symbolique semble ici prendre l'avantage, pourtant l'indifférence du *fuzzer* à toute structure peut être un bienfait. Les mutations aléatoire étant plus agressives, elles permettent d'élargir le parcours de l'arbre d'exécution [35, 39] en modifiant plusieurs octets de l'entrée à la fois. Testant des parties de l'entrée laisser de côté par l'exécution symbolique [18].

Un autre avantage non négligeable du *fuzzing* est sa capacité à faire varier la taille de l'entrée. Les exécuteurs symboliques dynamiques se contentent de récupérer des contraintes sur la valeurs des octets passés au programme. Ce qui limite la taille maximale de l'entrée symbolique à celle de l'entrée concrète initiale (la *graine*). De manière générale, sans information au préalable et spécifique au programme, il est difficile de définir des bornes exactes sur la taille des entrées.

Pour finir, l'exécution symbolique possèdent plusieurs limitations : nombres flottants, arithmétique des pointeurs *etc.* En pratique, lorsqu'un exécuteur est confronté à ce type de problème, celui-ci génère des tests erronés (certaines contraintes sont perdus) ou refuse tout simplement d'en générer. Le *fuzzing* ne connaît bien évidemment pas ce problème et il peut donc parfois faire office de remplaçant dans ces situations.

Pour toutes ces raisons, l'idée d'utiliser les deux techniques conjointement semble prometteuse. La perspective est que chaque outil puisse palier aux faiblesses de l'autre, permettant ainsi d'exercer des parties du code inaccessible si traiter séparément.

4.2 Objectifs

La motivation première est, ici, de reprendre les idées énoncées par Pak [39] dans sa thèse et de les mettre en place au sein d'un système basé sur des outils à l'état de l'art. Cette mise en pratique est, en effet, nécessaire afin de juger de l'efficacité des principes énoncés. Il existe plusieurs moyens pour s'assurer de cela :

- constater l'apport des deux outils face à un exemple jugé difficile pour les outils pris séparément. C'est-à-dire arriver à retracer l'origine du cas de tests et voir ses évolutions. Ceci requiert une analyse manuelle, cette méthode est donc limitée au programme de petite dimension.
- montrer une corrélation entre l'augmentation d'une ou des métriques et les échanges de cas de test entre les deux outils. Cette méthode est plus adaptée aux programmes de taille « réelle ».

Elle permet aussi de mettre à l'épreuve les outils choisis et de se confronter par la même occasion aux différentes limitations théoriques connues. C'est aussi un moyen de répondre à certaines interrogations pratiques telles que : l'ordre de grandeur pour le temps d'exécution d'une exécution symbolique ou le type de programmes facile à analyser pour chaque type d'outils.

4.3 Structure

Le schéma global présenté figure 4.1 fait apparaître la structure ainsi que le fonctionnement haut niveau de Fuzzs2e. L'explication suivante commence à gauche par l'exécuteur symbolique. La largeur des flèches indiquent la quantité relative de cas de test échangés entre les différents composants.

L'exécuteur symbolique réalise une exécution concolique à partir d'un ensemble de cas de tests candidats. Son exécution s'arrête si un des deux seuils présentés par Pak [39] est atteint. Le premier est le laps de temps d'exécution total. Le second est le nombre de chemins achevés. A la fin de son exécution, tous les états existants, même ceux n'ayant pas atteins la fin du programme, doivent générer un cas de test correspondant à leur *path condition* actuel. Ces entrées « partielles » sont ensuite passées au *fuzzer*.

Celui-ci applique ses mutations et autres heuristiques sur les entrées fournies en prenant garde de laisser intacte les octets dits « contraints ». Ces octets dont les valeurs sont les solutions fournies par le SMT solver assure que l'exécution concrète empruntera le chemin correspondant au *path condition* calculé par l'exécuteur symbolique.

Pour les entrées partielles, cela signifie que le *fuzzer* commence son *fuzzing* n'ont pas à la racine mais à partir de l'*état* où l'exécuteur symbolique s'est arrêté. Une fois « complétées » les entrées sont exécutées concrètement. Au cours de l'exécution plusieurs informations sont récupérées : le statut de sortie du programme ainsi que sa couverture de code (*basic blocks* ou branchements). La première information permet de trier les cas de test afin de mettre de côté ceux amenant à un arrêt brutal du programme. La couverture de code, quant à elle, est utilisée afin de choisir le prochain cas de test envoyé à l'exécuteur symbolique. La boucle continue ainsi jusqu'à l'épuisement des candidats ou lorsque l'utilisateur le désir.

Ce système est très similaire à celui de Pak à quelques exceptions près. Tout d'abord, ici le système contient deux points d'entrées il n'est pas nécessaire de commencer par l'exécuteur symbolique, le *fuzzer* peut très bien être aussi un point de départ. Ensuite, la méthode de génération des cas de test partielles est très différente. Ici, nous nous contentons d'utiliser uniquement la solution fournie par le SMT solver. C'est-à-dire que pour réexercer un chemin nous nous basons seulement sur un seul représentant de la classe. Pak propose de son côté de restreindre les valeurs possibles des octets contraints aux solutions du *path condition*. Cette énumération est très couteuse et très lente si nous nous reposons uniquement sur un SMT solver. Pak donne aussi une solution à ce problème dans sa thèse mais cela nous limite à une certaine catégorie de formule 3.3.

Avant d'aborder les modifications apportés aux deux logiciels, nous nous attardons maintenant sur les raisons qui nous ont fait choisir S2E et AFL.

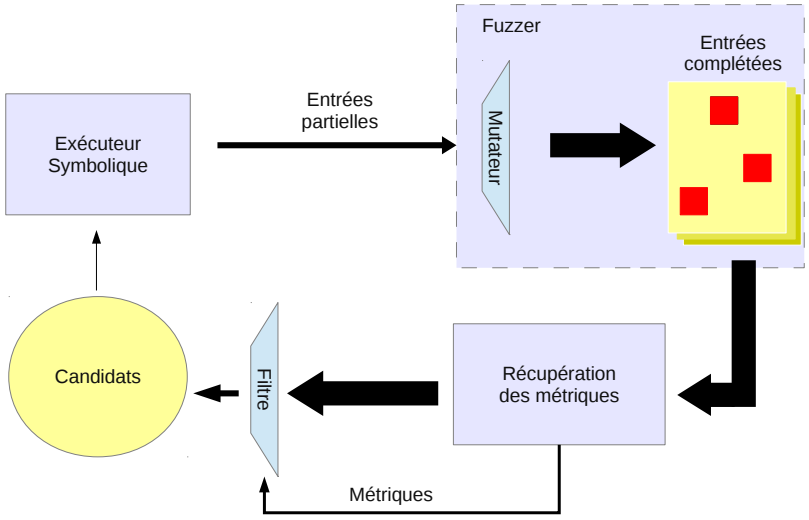


FIGURE 4.1: Schéma global du système.

S2E

Il n'existe pas actuellement de projet phare *open source* lorsque l'on s'intéresse au exécuteur symbolique traitant des binaires. Nous pouvons citer : Avalanche [31], Bitblaze [47], Fuzzgrind [16] et S2E. Parmi eux, S2E nous a semblé être le projet le plus proche de l'état de l'art et le plus mature comme le montre l'état de l'art de T. Chen et *al.* [19].

L'autre avantage de S2E est qu'il a été pensé comme un *framework*. Cela signifie qu'il est possible de construire des outils se basant sur lui sans devoir modifier son code interne car il a été prévu pour cela.

AFL

En ce qui concerne les *fuzzers*, le choix est relativement plus large. Des fonctionnalités avancées comme la possibilité de fournir des grammaires ou autres ne nous étant pas utile, la possibilité d'implémenter un *fuzzer* a été évoqué. Cependant, AFL possède plusieurs points forts en plus de sa renommé qui ont finis par nous convaincre.

AFL possède des heuristiques déterministes en plus de celles aléatoires habituelles. Il possède un ensemble de fonctionnalités et autres scripts utilitaires qui sont pratiques lorsqu'il est question de traiter plusieurs centaines de cas de tests comme par exemple le triage automatique des cas de tests par le statut de sortie du programme ou encore son *status screen* qui permet d'avoir un suivi de la session en cours. De plus ces performances sont très bonnes, nous avons pu constater des pics à plusieurs milliers d'exécution du programme cible par seconde. Pour finir, sa capacité à partager des cas de test entre plusieurs ins-

tances d'AFL a grandement simplifié notre implémentation de Fuzzs2e.

4.4 Modifications apportées aux deux logiciels

Nous avons nos deux composants principaux, il s'agit maintenant de les assembler afin de mettre en place Fuzzs2e.

Pour fonctionner, le système doit être capable de réaliser deux tâches principales. Premièrement, l'information à propos des octets contraints au sein d'une entrée doit pouvoir être transmise entre l'exécuteur symbolique et le *fuzzer*. La seconde tâche porte sur le partage d'information dans l'autre sens, elle consiste à choisir le « meilleur » cas de test pour la prochaine exécution symbolique.

Première tâche : Transmission des contraintes sur les octets

Le plugin `TestCaseGenerator` de S2E génère et affiche sur la sortie standard et au sein des journaux d'événements le cas de test correspondant au chemin venant d'être achevé. Bien que pratique par la vision qu'il apporte sur le déroulement de l'exécution dans l'état actuel, il n'est pas utilisable au sein de notre système.

Nous avons donc créer notre propre plugin de génération de cas de tests nommé `InputGenerator`. La première différence avec le plugin « officiel » de S2E et qu'il écrit chaque cas de test dans un fichier séparé en suivant la convention de nommage d'AFL. En effet, AFL est capable de récupérer et de traiter nos entrées grâce à sa fonction de synchronisation. Nous souhaitons maintenant partager l'information à propos des octets contraints pour chaque entrée. Pour ce faire, nous avons donc associé un fichier de meta-données à chaque cas de test. Celui contient une liste croissante d'indice correspondant aux octets ne devant pas être modifiés. La liste $[0, 3, 5, 7]$ signifie que l'entrée associée est d'au moins 8 octets et que le premier, le quatrième, le sixième et le huitième octet ont leur valeur fixée. Il suffit donc de modifier ensuite le plugin `InputGenerator` pour qu'il génère aussi les meta-données puis faire en sorte qu'AFL récupèrent ces fichiers et les prennent en compte lors de ses mutations. Mais avant tout cela, il faut définir les octets devant être contraints.

D'après ce que nous avons dit un peu plus haut, les octets contraints correspondent aux octets de la variable symbolique apparaissant dans la formule envoyée au SMT solver, c'est-à-dire le *path conditio*. En effet ces octets sont les seuls qui influent sur le flux de contrôle du programme et qui caractérisent donc le chemin pris par l'exécution. S2E ne peut fournir par défaut ce type d'information.

Cependant, il a la capacité de considérer séparément chaque octet composant une variable symbolique, car il les considère comme des tableaux d'octets. Le modèle de S2E apporte donc la vision nécessaire pour que nous puissions récupérer cette information. Pour ce faire, deux moyens existent. Le premier consiste à considérer les expressions composant le PC final. La façon dont S2E (Klee) construit ces expressions au cours de l'exécution symbolique ne nous permet pas de récupérer les octets contraints efficacement. Il faut donc *parser* la formule finale pour extraire cette information. Cela revient à parcourir tout l'Abstract Syntax Tree de la formule pour chercher les références aux

variables symboliques avant chaque génération d'un cas de test. Cette solution est donc coûteuse.

L'autre solution consiste à s'intéresser uniquement aux accès mémoires du programme. Si le programme lit en mémoire un octet symbolique celui-ci est susceptible d'influer le flux de contrôle du programme. Cette heuristique beaucoup plus légère assure toutefois que l'ensemble des indices récupérés est un sur-ensemble des octets influant véritablement sur le PC. Cette approche reste valide même si elle peut entrainer des faux positifs. De ce fait, le cas de test en question respecte bien les contraintes issues du *path condition*. Il se peut juste qu'il y en ait de superflu. Si le programme réalise une copie de l'entrée de l'utilisateur, tous les octets seront considérés comme devant être contraints même si en réalité un seul un octet influe réellement sur l'exécution. C'est cette approche qui est implémentée dans le plugin `InputGenerator` et avec ceci notre première tâche est résolue.

Deuxième tâche : choix du meilleur cas de test

La communication est assurée de l'exécuteur symbolique vers le *fuzzer*. Nous nous intéressons maintenant à un moyen de partager les avancées d'AFL avec S2E. Au cours de son analyse, AFL mute, exécute puis trie les cas de tests. Il stocke les cas les plus intéressants, à son sens, dans une file. Nous souhaitons récupérer cette file et extraire un seul cas. La notion de cas de test « intéressant » est floue. Nous définirons, ici, comme intéressant une entrée faisant augmenter la couverture de code totale de l'analyse en cours. Cela signifie que si l'entrée en question passe par un *basic block* ou un branchement qui n'avait jusqu'au alors jamais été exécuté, nous le mettons dans une file de candidats. Plus la nouvelle couverture apportée par le candidat est importante est plus ce candidat est traité en priorité.

Fuzzs2e procède donc de cette façon. A la fin d'une exécution symbolique, il applique l'heuristique précédente aux entrées d'AFL afin de mettre à jour la file de priorité des candidats. Puis, il récupère le premier élément de la file et l'utilise en tant que *graine* pour une nouvelle exécution symbolique.

Importance des modifications

Les modifications apportées à S2E représente 500 lignes de codes C++. L'implémentation de la récupération de la liste des candidats depuis la file d'AFL et de l'heuristique `TotalCoverage` tient en 380 lignes de Python. Enfin les modifications faites au sein d'AFL pour intégrer les méta-données et les contraintes sur les octets correspondent à environ une centaine de lignes de code C. Ces résultats ont été obtenus à l'aide du programme sloccount [1].

1. `http://www.dwheeler.com/sloccount/`

CHAPITRE 5

Expérimentations

5.1 Bzip2

Le programme de compression et décompression bzip2 [46] et sa bibliothèque associée représente une cible intéressante pour tester Fuzzs2e. Tout d'abord, il rentre dans la catégorie des programmes pouvant être tester facilement à la fois par S2E et AFL. Nous nous concentrons plus particulièrement sur la version 1.0.5 qui possède un bug connu répertorié par la CVE-2010-0405 : un *integer overflow* lors de la décompression d'une archive valide.

Résultats

Pour évaluer nos tests nous nous basons sur 3 métriques : le nombre de chemins « uniques », la couverture des basic blocks ainsi que la couverture des branchements (*edges*). Enfin, une autre métrique importante pour s'assurer du bon fonctionnement de Fuzzs2e et le nombre d'entrées importées par AFL. Ce nombre indique la quantité de cas de test produits par S2E et jugés intéressants par AFL. Cette métrique montre donc l'importance des échanges entre les deux outils.

Nous avons testé la fonction de décompression de bzip2 sur l'ensemble des outils : AFL, S2E et Fuzzs2e. Un premier test avec AFL sur une période de 4 heures affiche un début de stagnation : au bout d'une heure pour le nombre de chemins uniques et au bout de seulement 20 minutes pour la couverture de code, comme le montre les graphiques présents en annexe (Figure C.1 & Figure C.2). De plus, le bug n'est pas déclenché. La même expérience réalisé avec S2E affiche des résultats légèrement moins bons aux niveaux des métriques et aucun crash n'est trouvé. L'exécution symbolique étant plus coûteuse, il semble normal qu'AFL possède un départ plus rapide quitte à être rattrapée par la suite.

En renouvelant ces tests, nous obtenons des résultats quasi similaires pour ce même laps de temps. Les deux outils atteignent une phase de seuil dès la première demi-heure. Des tests d'une heure maximum sont donc suffisants pour voir apparaître des potentielles évolutions.

A partir de là plusieurs scénarios sont envisageables pour Fuzzs2e. Le premier scénario que nous ayons considéré se déroule de la façon suivante. Nous commençons par récupérer les contraintes sur les octets de la *graîne* avant de l'utiliser avec Fuzzs2e. De cette manière, le *fuzzer* ne travail uniquement que sur des entrées contraintes. L'idée étant de concentrer l'activité du *fuzzer* sur les sous-arbres définis par les entrées partielles issues de l'exécution symbolique. Avec ce scénario les résultats obtenus sont légèrement moins bons que les résultats initiaux par contre le nombre de cas de test importés par AFL dépasse la cinquantaine pour un nombre total de chemins proches des 900. L'ordre de grandeur de la perte de couverture est de quelques dizaines de *basic blocks* ou d'*edges* en moins. Appliquer des contraintes au *fuzzer* dès le départ restreint le *fuzzer* au seul sous-arbre préalablement découvert par S2E, ce qui limite l'apport d'AFL.

Les prochains tests réalisés se base sur des *graînes* non contraintes. Les résultats obtenus en termes de couverture sont équivalents à ceux d'AFL. Par contre, le nombre de cas de tests importé chute à moins d'une dizaine. L'exécuteur symbolique ne semble donc pas avoir beaucoup d'impacte durant la campagne de test.

Afin de découvrir la raison de ces résultats, nous avons fait varier les différents paramètres influant sur le comportement de Fuzzs2e : La valeur des seuils de l'exécution symbolique (temps imparti et nombre des chemins exécutés) et la couverture de code utilisée par l'heuristique *TotalCoverage* (*basic block* ou *edges*). Malgré toutes ces variations, les résultats obtenus restent semblables à ceux mentionnés précédemment.

A ce stade, deux causes semblent plausibles pour expliquer ces résultats. Soit le concept en lui-même est inefficace ou soit la cause réside au sein du programme testé.

Analyse de la difficulté

Au vue des résultats pour le logiciel bzip2, il semble pertinent d'essayer de mettre à jour de façon plus précise l'origine des difficultés rencontrées par S2E et AFL. Pour ce faire, nous avons fabriqué des programmes simplifiés mettant en évidence les différents comportements et limitations de S2E ou d'AFL. Le but étant de faciliter l'analyse manuelle des résultats générés par les deux outils.

Les cas de test produit par Fuzzs2e pour bzip2 possèdent quasiment tous une somme de contrôle ou *checksum* erronée. Pour être plus précis une archive bzip2 possèdent au minimum deux *checksums*. Cette somme codés sur 4 octets permet de s'assurer de l'intégrité d'un bloc de l'archive ou de son ensemble. Les 4 octets de *checksum* dépendent donc des autres octets de l'archives. AFL n'est évidemment pas capable de détecter ces dépendances et il existe donc une infime chance que celui-ci affecte les bonnes valeurs aux bons 4 octets correspondant au *checksum*. A l'inverse S2E doit voir apparaître ces dépendances lors de l'appel à la fonction de hachage (ici CRC32) avec des arguments symboliques.

Pour étudier ce phénomène nous nous baserons sur l'exécution symbolique du programme crc32 dont le code source est présenté en annexe (Listing A.1). Ce programme très simple accepte une entrée binaire composée d'un identifiant ou « *magic number* » (`BOB` en ASCII), de données (au moins

1 octet) et d'un *checksum* CRC32 portant sur les données. Le logiciel s'arrête brutalement lorsque l'entrée est valide : en-tête, longueur et *checksum*. Cet exemple est ensuite analysé par AFL et S2E séparément avec pour seul cas de test initial la chaîne de caractères formé de 20 « *A* ». Comme cet exemple est simple quelques minutes suffisent pour s'assurer des difficultés rencontrés par les deux outils.

La présence d'un en-tête dans cet exemple sert à jauger de l'efficacité de l'heuristique présente au sein d'AFL servant à découvrir les « *magic number* ». Celle-ci est efficace car après seulement quelques secondes AFL favorisent les cas de tests comportant cet en-tête.

En ce qui concerne S2E, ce type d'en-tête est trivial et ne présente pas de difficulté. Les deux outils sont donc capables de générer des cas de test pour les chemins terminant par *EXIT_BAD_SIZE*, *EXIT_BAD_ID* et *EXIT_BAD_CRC* mais tous les deux n'arrivent pas à produire une entrée valide. Sans surprise, les heuristiques d'AFL ne fonctionnent pas dans ce cas. Il a recours aux tests aléatoires pour espérer trouver le *checksum* en question sans grand succès. La probabilité que celui-ci fabrique une entrée valide est difficile à calculer car elle dépend des mutations apportées. Dans le meilleur des cas où AFL se base sur une entrée correcte à l'exception du *checksum*, la probabilité de générer le bon *checksum* n'est que de $1/2^{32}$. Les difficultés éprouvées par AFL ici ne sont donc pas étonnantes.

Il reste à identifier le comportement de S2E face à la contrainte issue de la comparaison du *checksum* calculé par le programme et de celui présent au sein de l'entrée. D'après les résultats, S2E semble être bloqué. Il « *fork* » toujours à la même adresse et semble réaliser une recherche exhaustive sur les valeurs possibles du *checksum*. Seul un octet diffère entre chaque cas de test générés. Avec de la patience, S2E finirait par trouver le bon *checksum* parmi les 2^{32} possibilités. Ce comportement est étrange. A l'exception de la bonne valeur, le reste des cas de tests qui peuvent être énumérés engendrent le même chemin au sein de l'arbre d'exécution, à savoir celui terminant par EXIT_BAD_CRC. Ceci est contraire au principe de l'exécution symbolique exerçant un chemin une seule et unique fois. Ce comportement est donc une particularité de S2E.

Nous nous intéressons maintenant à l'origine de ce comportement, intuitivement deux causes semblent plausibles : les limites du SMT solver sont atteintes ou alors les formules construites par S2E sont incorrectes. L'étude des requêtes au SMT solver permet d'avoir une meilleur compréhension du fonctionnement de S2E à ce moment là. Considérons l'extrait de requête présentée en annexe (Listing B.1) écrit en KQuery [1]. Les lignes de commentaires commençant par # traduisent les contraintes en langage C. Les trois premières correspondent à l'identifiant puis les suivantes fixent les octets de l'entrée à 0 à l'exception du dernier qui correspond à l'octet de poids fort du *checksum*. Plusieurs contraintes influent sur sa valeur. Tout d'abord une liste de valeurs lui sont refusées (ligne 45), ces valeurs correspondent aux tests déjà générés, l'énumération semble donc être le comportement « normal » de S2E.

Concentrons nous sur la contrainte appliquée au dernier octet de l'entrée (ligne 31) :

```
buffer[19] & 0xf == 0xa
```

1. La documentation expliquant le fonctionnement et la syntaxe du langage est disponible sur le site `https://klee.github.io/docs/kquery/`

Le *checksum* dépend de la valeur des octets des données par l'utilisateur. Pourtant celui-ci est associé à une constante plutôt qu'une expression symbolique mettant en relation le champ de donnée de l'entrée (`buffer[i]` avec $3 < i < 16$). Cela semble être le signe d'une concrétisation.

Le moteur d'exécution confronter à un des problèmes difficiles pour la DSE aurait donc simplifié la formule faisant intervenir les variables symboliques liées aux données. Nous avons ici qu'une hypothèse car S2E ne laisse aucune trace permettant de confirmer la présence de ces concrétisations.

Malgré tout il est intéressant d'essayer d'identifier les potentielles causes de ces simplifications au sein même du code source de notre application. Pour cela, nous nous intéressons aux implémentations de CRC32 au sein de la bibliothèque zlib [25] utilisé par le programme crc32 et au sein de bzip2. Les deux implémentations utilisent une table de hachés pré-calculées comme le montre les Listings A.2 et A.3. Afin de retourner un *checksum*, cette table est indexée à l'aide d'une valeur dérivée des variables symboliques (les variables `buf` et `cha`). Visiblement lors de l'exécution symbolique de ce code, S2E est confronté au problème de l'arithmétique des pointeurs 1.5 ce qui expliquerait la concrétisation.

Le SMT solver n'est décidément pas fautif, ici, car il n'y a aucune trace de requêtes dont la résolution excèdent le temps imparti au sein des journaux d'événements de S2E ou de ralentissements qui pourraient ce genre de requête trop longue.

5.2 Cyber Grand Challenge (CROMU004) : pcm_server

Nous avons appliqués Fuzzs2e au challenge CROMU004 [2] du Cyber Grand Challenge. Le programme pcm_server accepte sur l'entrée standard des fichiers audio au format PCM et décodent le message en morse qu'ils contiennent. Il possède plusieurs caractéristiques intéressantes. Tout d'abord, il ne contient pas d'instructions ou de fonctions complexes susceptibles de perturber S2E. Enfin la vulnérabilité expliquée dans la description du challenge semble être un cas favorable pour Fuzzs2e. C'est un *buffer-overflow* qui ne peut être déclenché que s'il existe une relation particulière entre deux entiers extraits de l'en-tête du fichier fourni à pcm_server. Nous espérons donc que S2E soit capable de trouver cette relation puis qu'il la transmettre à AFL. Celui n'a alors plus qu'à faire varier la taille de l'entrée jusqu'à atteindre un crash.

Résultats

La première analyse de pcm_server par Fuzzs2e s'est avérée être concluante. Au bout d'environ 30 minutes, AFL affiche un crash. Pour juger de l'apport de Fuzzs2e dans la découverte du bug, nous nous assurons que les outils seuls ne soient pas suffisant pour trouver le bug. AFL ne trouve rien après plus d'une heure d'analyse et S2E, lui, achève son analyse symbolique en moins de cinq minutes mais les cas de tests produit ne mettent pas en évidence le bug. Nous nous retournons alors vers Fuzzs2e afin de nous assurer que le cas de test déclenchant le bug est bien issu de la collaboration des deux outils.

2. `https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges/CROMU_00004`

La convention de nommage d'AFL faisant apparaître le cas de tests source d'où provient l'entrée mutée, il est ainsi possible de reconstruire l'historique de mutation d'une entrée. En étudiant l'historique du cas de tests en question, nous nous apercevons qu'il est issu uniquement d'entrées produites par AFL. Apparemment, l'interaction entre les deux outils ne semble pas être à l'origine de la découverte du bug ce qui semble contredire nos précédents tests. Nous analysons de nouveau pcm_server avec AFL, cette fois ci AFL déclenche le bug en 15 minutes. Plusieurs autres analyses avec AFL confirment le caractère aléatoire de cette découverte. Ce n'est donc finalement qu'une histoire de chance, non seulement AFL a lui seul suffit pour trouver le bug mais étrangement S2E semble passer totalement à côté.

Le programme pcm_server pose donc problème à S2E. Par la suite, nous cherchons à comprendre l'origine de cette difficulté.

Mise en évidence de la difficulté

Nous allons maintenant étudier de plus près pcm_server afin de comprendre ces résultats contre intuitifs. Premièrement nous allons présenter plus en détail le bug et son origine. Pour déclencher ce bug le fichier doit posséder un en-tête valide, celui se compose de 12 octets obligatoires :

- les 4 premiers octets sont le « *magic number* » est doivent correspondre à la chaîne de caractères « `PCM␣` ».
- Un entier non signé correspondant à la taille des données.
- Un entier non signé correspondant au nombre d'échantillons qui sont en réalité des blocs de 16 bits.

Étudions maintenant le code source de pcm_server s'occupant de vérifier cet en-tête (Listing 5.1).

Listing 5.1: Vérification de l'en-tête d'un fichier PCM.

```
#define MAX_PCM_SIZE 1048576
#define MAX_SAMPLES 524282
...

/* use our pcm_header struct to check some input values */
p = (struct pcm_header *)pcm;
if (p->ID != PCM) {
  puts("Invalid PCM format\n");
  exit(INVALID_PCM_FMT);
}
if (p->NumSamples > MAX_SAMPLES) {
  puts("Invalid PCM length\n");
  exit(INVALID_PCM_LEN);
}
if (p->NumSamples == 0) {
  puts("Invalid PCM format\n");
  exit(INVALID_PCM_FMT);
}

/* make sure the SampleSize value is valid */
if (p->DataSize*8 / p->NumSamples != 16) {
  puts("Invalid PCM length\n");
```

```
23   exit(INVALID_PCM_LEN);
24 }
```

Le test ligne 17 s'assure que la taille en bits des données soit bien un multiple de 16 et c'est d'ailleurs le seul et unique test sur la taille de l'entrée ! A aucun moment, la taille de l'entrée fournit par l'utilisateur n'est comparée avec `MAX_PCM_SIZE` qui est la taille du tampon `p` alloué sur la pile d'exécution. Le bug provient en réalité du type de l'expression `p->DataSize*8 / p->NumSamples`, elle retourne un entier alors que le programmeur semble espèrer en réalité un nombre flottant (un réel). Le test vérifie seulement le quotient de la division euclidienne. Par conséquent, le couple de valeur $(23, 11)$ vérifient bien la condition. Pourtant $23 \times 8 = 184$ n'est clairement pas un multiple de 16, $184/16 = 16.72$. Pour mettre à jour le bug, il faut donc que nos outils soient capables de générer une entrée satisfaisant la formule suivante dans la logique *BitVector* avec x et y deux *bitvectors* de 32 bits.

$$y \leq \texttt{MAX\_SAMPLES} \wedge x > \texttt{MAX\_PCM\_SIZE} \wedge x \times 8/y = 16$$

Avec ces informations nous sommes donc capables de construire une entrée de 12 octets provoquant l'arrêt brutal de pcm_server. De plus ce type de formule est à la portée des SMT solvers actuels. Nous devons donc nous intéresser à la nature des requêtes envoyées par S2E.

Un extrait de ces requêtes est présenté en annexe (Listing B.2). Nous avons enlevés délibérément les contraintes concernant pas la vérification de l'entête pour plus de clarté. Naturellement, toutes les contraintes liées à l'entête y sont présentes : l'identifiant, le nombre d'échantillons et son rapport avec la taille des données. Cependant, la contrainte liée à la taille maximale du tampon `pcm` ($x >$ `MAX_PCM_SIZE`) n'apparaît nulle part. Or, c'est justement cette contrainte qui assure un débordement. Avec du recul, cela s'avère être normal. Cette contrainte est implicite, elle n'apparaît en aucune façon dans le flot de contrôle du programme. L'exécution symbolique n'est pas capable à elle seule d'inférer ce type d'information.

5.3 Démonstration des capacités du système

Les exemples choisis jusqu'à présent ne mettent en évidence que les limites de FuzzS2E. Afin de s'assurer de l'intérêt d'un tel système, nous allons étudier un programme fabriqué par nos soins nommé parity. Ce programme, dont le code source est présenté en annexe (Listing A.4), a la particularité de mettre en échec AFL et S2E mais est résolu facilement par Fuzzs2e.

Le programme parity suit le même schéma que le programme crc32. Il accepte une entrée binaire suivant le format suivant :

- Un « *magic number* » qui correspond à la chaîne de caractères `ALICE` en ASCII.
- Des données : Au minimum deux nombres flottants de 4 octets chacun (`float`) doivent être présents.
- un « *checksum* » de 4 octets : celui est construit à partir du nombre arbitraire `0x76543210`. Son bit de poids le plus faible est réservé pour stocker la parité [3] des données.

3. La parité d'une donnée binaire est le nombre de bits valant 1 le tout modulo 2

Le programme s'arrête brutalement s'il reçoit une entrée valide au sens où celle-ci rempli les conditions suivantes : en-tête correct, *checksum* correct, une parité des données valant 1 et afin la fonction `trial` prenant en arguments les deux nombres flottants doit retourner 1.

La première condition est remplie sans trop d'effort par S2E et AFL. Trouver la seconde pose plus de problèmes à AFL car comme avec le programme crc32 les probabilités de tomber sur le bon *checksum* sont faibles. Le calcul de parité est trivial pour S2E, il arrive donc sans soucis à reconstruire le *checksum* correspondants aux données. La fonction `trial` représente une barrière infranchissable pour S2E, cependant. Elle comporte des instructions sur des nombres flottants symboliques. Or S2E ne sait pas gérer ce type d'instructions et abandonnent l'*état* en question sans même générer de cas de test.

La fonction `trial` est très simple en réalité. Seul les octets de poids faible des deux flottants sont pris en compte, par conséquent le nombre de solution est suffisamment grand pour que le *fuzzing* soit efficace. Malheureusement, AFL bloque dès la seconde condition et il n'arrive donc pas à exercer d'autres chemins à part ceux terminant par EXIT_BAD_SIZE, EXIT_BAD_ID et EXIT_BAD_CRC. De son côte S2E arrive à exercer deux autres chemins : EXIT_BAD_PAR et EXIT_FAILURE mais n'arrive tout de même pas à aboutir au crash.

Les deux outils bloquent à des endroits différents et il est facile de voir qu'en les combinant le crash serait trouvé facilement. Plus explicitement, il suffirait de « *fuzzer* » les deux nombres flottants après avoir définit l'en-tête et le *checksum*. C'est-à-dire appliquer AFL sur un des cas de test produit par S2E, c'est exactement ce que permet Fuzzs2e.

En pratique Fuzzs2e mets à peine une trentaines de secondes pour déclencher ce bug, là où les deux outils pris séparément sont impuissants. La capture d'écran présentée en annexe (Figure C.2) montre un des cas de tests menant à un crash ainsi que son historique de mutation.

Conclusion

Bilan

Les résultats à l'issue des différentes expérimentations sont insuffisants. L'apport de Fuzzs2e n'a pu être constater que sur un programme construit dans ce seul but. Une phase d'expérimentation plus longue et plus variée apporterait une meilleur vision des capacités de l'outil : différents programmes et différents type de bugs.

Bzip2 et pcm_server sont deux contre-exemples mettant en avant différentes faiblesses de Fuzzs2e. Pour bzip2, après réflexion, les mauvais résultats obtenus ne sont pas étonnant. Les fonctions de hachages représentent une difficulté à la fois pour l'exécution symbolique et le *fuzzing*. Le logiciel pcm_server quant à lui a permis de révéler une faiblesse du concept même.

Fixer la valeur d'un octet lors du *fuzzing* peut dans certains cas entraver le travail du *fuzzer*, en l'empêchant de modifier les octets de l'entrée susceptibles de déclencher le bug. Non seulement, l'exécuteur symbolique est incapable de trouver le bug mais cette limite se répercute aussi sur le *fuzzer*.

Ceci semble apporter plus de crédit à l'approche initiale de B. Pak, qui lui fait varier les valeurs des octets contraints au sein l'ensemble solution définit par le *path condition*. La solution consistant à faire partager simplement les cas de tests entre les deux outils, sans qu'il y ait de notions d'octets contraints, ne semble pas dépourvu d'intérêt pour autant. L'accent est mis, dans ce cas, sur l'heuristique chargée de choisir le cas de test à muter ou à exécuter symboliquement.

Du point de vue des objectifs secondaires, la réalisation de Fuzzs2e a permis de nous confronter aux deux outils : leur utilisation, leur fonctionnement interne et leurs limites. Prenons le cas de S2E, ces performances sont très variables, elles dépendent énormément du programme cible et des chemins parcourus. Plus le nombre d'instructions agissant sur des variables symboliques est grand et plus l'exécution est longue. Pour avoir une vision plus concrète, l'exécution concolique de bzip2 sur une archive de 176 octets met plus de 15 minutes pour produire le premier cas de tests. A l'inverse, l'exécution symbolique totale du programme parity ne varie que très peu selon la taille de l'entrée. Ceci semble normal car parity ne considère en réalité que les 12 premiers octets.

En ce qui concerne AFL, il possède deux limites fortes à nos yeux. A part la sortie standard, il est incapable de gérer d'autres vecteurs d'entrée. Il est aussi limité à un seul type de programme prenant une unique entrée avant de produire un résultat et de s'arrêter. Pour tous les programmes comportant

des états internes à la manière d'un serveur web ou d'un programme comportant une interface graphique, AFL n'est plus adapté. Ces limitent sont ancrées au sein de la conception même d'AFL et sont étroitement liées à ces performances.

Ouverture et perspectives futures

Dans cette section nous discutons de quelques pistes d'améliorations possibles.

Instrumentation partagée

Fuzzs2e souffre de deux gros problèmes l'un affectant ces performances et l'un affectant potentiellement sa précision.

Actuellement chaque nouvelle exécution symbolique correspondant à un nouveau processus indépendant de S2E. Cela signifie que la nouvelle instance recommence tout depuis le début et réexécute symboliquement des portions de code déjà traité par les anciennes instances, ce qui est inefficient.

L'autre problème est lié aux deux environnements dans lequel est testé le programme cible. Aujourd'hui, S2E teste le programme dans une machine virtuelle hébergeant une debian 32 bits tandis qu'AFL teste celui-ci sur la machine hôte, qui au moment des tests était une debian 64 bits. Malgré les précautions prises pour faire coïncider les versions des distributions, les options de compilations (-m32) *etc.* le moindre changement de version entre deux bibliothèques partagées peut être préjudiciable.

Partagé l'instrumentation entre les deux outils permettrait de régler ces deux problèmes d'un seul coup. S2E utilise la capacité de qemu à prendre des *snapshots* du système pour sauvegarder le contexte concret de ces *états*. Il est possible alors de relancer le *fuzzing* depuis ces *états*/*snapshots*, ce qui s'apparente aux travaux de Majumdar et Sen [35]. Ceci réglerait le premier problème, de plus comme les deux outils travaillerait sur la même machine virtuelle le second disparaitrait lui aussi.

Cette solution nous était connue depuis le départ. Cependant cette approche nécessite un temps de développement plus important.

Heuristique de sélection des cas de test candidats

Actuellement Fuzzs2e repose sur trois heuristiques distinctes afin de choisir le prochain cas de test ou chemin à parcourir : le parcours en profondeur de S2E, l'optimisation du *edge coverage* à la façon d'AFL et enfin la file de priorité des candidats propre à Fuzzs2e. La combinaison d'heuristiques peut être une bonne chose comme le démontre Cadar et *al.* [14], mais seulement si celle-ci est maitrisée.

Découpler la partie s'occupant des heuristiques et l'a partagé entre les outils, permettrait une plus grande souplesse. Il serait alors possible de tester des combinaisons plus poussées et même de changer d'heuristique au cours de l'exécution pour un outil ou voire pour tous. Il serait intéressant aussi de s'attarder sur les techniques d'*Adaptive Random Testing*. L'état de l'art proposé au sein de l'article de Anand et al. [1] montre tout un ensemble de nouvelles

approches différentes de celles proposées ici et ne se limitant pas à l'optimisation de la couverture de code.

Recherche de vulnérabilités

AFL et S2E sont deux outils qui ont vocation à rechercher des bugs pouvant remettre en cause la sécurité du programme et du système sur lequel il s'exécute. Pourtant, il ne semble pas pertinent de les considérer comme des outils recherchant des *vulnérabilités*. Par défaut, aucun de ces outils ni les techniques sur lesquels ils reposent (l'exécution symbolique et le *fuzzing*) ne recherchent spécifiquement des *buffer overflow* ou des *use-after-free* au sein du programme.

Ils réalisent « simplement » des tests. Leur but consiste à exécuter, à exhiber le plus grand nombre de comportements du programme. En réalité, ces types d'outils ne recherchent donc rien en particulier. Ils sont cependant un socle nécessaire afin que des analyses spécifiques plus poussées puissent être appliquées de manière dynamique sur un programme. Le programme pcm_server et l'incapacité de S2E à inférer la taille maximale du tampon est un exemple flagrant. Des exemples d'analyses dédiés à un type de bugs spécifiques existent à la fois dans un contexte statique [23] et à la fois dans un contexte dynamique. Par exemple IntScope [49] se base sur une exécution symbolique et une analyse de teinte pour détecter des *integer overflow*.

La généralisation de ce type d'analyse ou l'utilisation de connaissances spécifiques à un type de bug pour diriger la campagne de tests, sont des perspectives intéressantes qui à notre connaissance non pas encore été explorées.

Annexes

ANNEXE A

Codes sources

A.1 Etude de l'origine des difficultés de Fuzzs2e

Bzip2

Listing A.1: Programme crc32

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <arpa/inet.h>

#include <zlib.h>

#define MAX_SIZE 100
#define HEADER_SIZE 3
#define HEADER "BOB"

enum exit_status {
  EXIT_BAD_SIZE = 2,
  EXIT_BAD_ID
  EXIT_BAD_CRC
};

int main(void)
{
  unsigned char buffer[MAX_SIZE];

  int c;
  int i = 0;
  unsigned real_cksum, in_cksum;
  unsigned size = 0;

  while ((c = getchar()) != EOF && size < MAX_SIZE) {
```

```
30    buffer[size++] = (unsigned char) c;
31   }
32
33   if (size < HEADER_SIZE + 4 + 1)
34    return EXIT_BAD_SIZE;
35
36   for (; i < HEADER_SIZE; ++i)
37    if (buffer[i] != HEADER[i])
38     return EXIT_BAD_ID;
39
40   printf("Size: %d\n", size);
41
42   for (int i = 0; i < size; ++i)
43    printf("%d: 0x%x %c\n", i, buffer[i], buffer[i]);
44
45   assert(i == 3);
46   real_cksum = ntohl(crc32(0L, &buffer[i], size - i - 4));
47   in_cksum = *((unsigned *) &buffer[size - 4]);
48
49   printf("Real checksum: 0x%08x\n", real_cksum);
50   printf("Input checksum: 0x%08x\n", in_cksum);
51
52   if (real_cksum != in_cksum) {
53    return EXIT_BAD_CRC;
54   }
55   else {
56    /* SEGFAULT */
57    int *p = 0;
58    return *p;
59   }
60
61   return EXIT_SUCCESS;
62 }
```

Listing A.2: Code source de zlib

```
1 /*
      ========================================================================
      */
2 #define DO1 crc = crc_table[0][((int)crc ^ (*buf++)) & 0xff]
      ^ (crc >> 8)
3 #define DO8 DO1; DO1; DO1; DO1; DO1; DO1; DO1; DO1
4
5 /*
      ========================================================================
      */
6 unsigned long ZEXPORT crc32(crc, buf, len)
7   unsigned long crc;
8   const unsigned char FAR *buf;
9   uInt len;
10 {
11   if (buf == Z_NULL) return 0UL;
12
13   ...
```

```
14
15   crc = crc ^ 0xffffffffUL;
16   while (len >= 8) {
17     DO8;
18     len -= 8;
19   }
20   if (len) do {
21     DO1;
22   } while (--len);
23   return crc ^ 0xffffffffUL;
24 }
```

Listing A.3: Code source de bzip2

```
1 /*-- Stuff for doing CRCs. --*/
2
3 extern UInt32 BZ2_crc32Table[256];
4
5 #define BZ_INITIALISE_CRC(crcVar)   \
6 {                                   \
7   crcVar = 0xffffffffL;             \
8 }
9
10 #define BZ_FINALISE_CRC(crcVar)     \
11 {                                   \
12   crcVar = ~(crcVar);               \
13 }
14
15 #define BZ_UPDATE_CRC(crcVar,cha)   \
16 {                                   \
17   crcVar = (crcVar << 8) ^          \
18          BZ2_crc32Table[(crcVar >> 24) ^ \
19                         ((UChar)cha)]; \
20 }
```

A.2 Mise en évidence des capacités de Fuzzs2e

Listing A.4: Code source de parity

```
1 #include <assert.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define MAX_SIZE 50
6 #define HEADER_SIZE 5
7 #define HEADER "ALICE"
8 #define MASK 0x76543210
9
10 enum exit_status {
11   EXIT_BAD_SIZE = 2,
12   EXIT_BAD_ID,
13   EXIT_BAD_CRC,
```

```
14   EXIT_BAD_PAR
15 };
16
17
18 int trial(float x, float y)
19 {
20   float res = y*y;
21
22   char x_p = *(char *) &x;
23   char y_p = *(char *) &res;
24
25   return x_p == y_p;
26 }
27
28 int success(void)
29 {
30   puts("OK");
31   return EXIT_SUCCESS;
32 }
33
34 int failure(void)
35 {
36   puts("BAD");
37   return EXIT_FAILURE;
38 }
39
40
41 int check_parity(unsigned char *buf, unsigned len)
42 {
43   int par = 0;
44
45   if (len < 1)
46     return -1;
47
48   for (int i = 0; i < len; ++i)
49     par += __builtin_parity(buf[i]);
50
51   return par % 2;
52 }
53
54 int main(void)
55 {
56   unsigned char buffer[MAX_SIZE];
57
58   float index1, index2;
59
60   int c, res;
61   int i = 0;
62   int size = 0;
63   unsigned real_par, in_par;
64
65   while ((c = getchar()) != EOF && size < MAX_SIZE)
66     buffer[size++] = (unsigned char) c;
67
```

```
68  /* HEADER + INDEX1 + INDEX2 + parity byte */
69  if (size < HEADER_SIZE + 2 * sizeof(float) +
        sizeof(unsigned))
70    return EXIT_BAD_SIZE;
71
72  for (; i < HEADER_SIZE; ++i)
73    if (buffer[i] != HEADER[i])
74      return EXIT_BAD_ID;
75
76  assert(i == HEADER_SIZE);
77  real_par = check_parity(&buffer[i], size - i -
        sizeof(unsigned));
78  real_par |= MASK;
79  in_par = *((unsigned *) &buffer[size - sizeof(unsigned)]);
80
81  /* Here AFL should be stuck (at least few minutes/hours):
82     There is 2^32 possible values for in_par.
83     However it's still easy for S2E. */
84  if (real_par == -1 || real_par != in_par)
85    return EXIT_BAD_CRC;
86
87  index1 = *((float *) &buffer[i]);
88  index2 = *((float *) &buffer[i + sizeof(float)]);
89
90  if (real_par == (0x1 | MASK))
91    return EXIT_BAD_PAR;
92
93  if (trial(index1, index2)) {
94    res = success();
95    int *p = 0;
96    return *p;
97  }
98  else
99    res = failure();
100
101 return res;
102 }
```

ANNEXE B

Extraits de journaux d'événements

B.1 Bzip2

Listing B.1: Exemple de requête KQuery effectuée par S2E durant l'analyse de crc32.

```
(query [
  (Eq (w8 0x42)
    (Read w8 0x0 v0___symfile___input0___0_1_symfile___0))
  # buffer[0] == 'B'

  (Eq (w8 0x4f)
    (Read w8 0x1 v0___symfile___input0___0_1_symfile___0))
  # buffer[1] == 'O'

  (Eq (w8 0x42)
    (Read w8 0x2 v0___symfile___input0___0_1_symfile___0))
  # buffer[2] == 'B'

  (Eq (w32 0x0)
    (And w32 N0:(Concat w32 (w8 0x0)
                (Concat w24 (w8 0x0)
                  (Concat w16 (w8 0x0)
                    (Read w8 0x3
                        v0___symfile___input0___0_1_symfile___0))))
           (w32 0xf)))
  # buffer[3] & 0xf == 0

  ...

  # Contrainte sur le dernier octet du checksum
  (Eq (w32 0xa)
    (And w32 N16:(Concat w32 (w8 0x0)
```

```
27                (Concat w24 (w8 0x0)
28                  (Concat w16 (w8 0x0)
29                    (Read w8 0x13
                          v0___symfile___input0___0_1_symfile___0))))
30           (w32 0xf)))
31   # buffer[19] & 0xf == 0xa
32
33   (Eq false
34     (Eq (w32 0x0) N17:(LShr w32 N16 (w32 0x4))))
35   # (buffer[19] << 4) != 0
36
37   (Eq false
38     (Eq (w32 0x1) N18:(And w32 N17 (w32 0xf))))
39
40   # N18 = (buffer[19] << 4) & 0xf
41   # N18 != 1
42
43
44   # Blacklist
45   (Eq false (Eq (w32 0x2) N18))
46   (Eq false (Eq (w32 0x3) N18))
47   (Eq false (Eq (w32 0xb) N18))
48   (Eq false (Eq (w32 0x4) N18))
49   (Eq false (Eq (w32 0x6) N18))
50   (Eq false (Eq (w32 0x5) N18))
51   (Eq false (Eq (w32 0x7) N18))
52   (Eq false (Eq (w32 0x8) N18))
53   (Eq false (Eq (w32 0xa) N18))
54   (Eq false (Eq (w32 0xc) N18))
55   (Eq false (Eq (w32 0xe) N18))
56   (Eq false (Eq (w32 0x9) N18))]
57
58 #  OK -- Elapsed: 0.004237
59 #  Solvable: true
60 #  v0___symfile___input0___0_1_symfile___0 =
61 #    [66,79,66,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,250]
```

B.2 pcm_server

Listing B.2: Extrait d'une requête KQuery effectuée par S2E durant l'analyse de parity.

```
1 array v0___symfile___input0___0_1_symfile___0[39] : w32 ->
      w8 = symbolic
2 (query [
3    (Eq (w32 0x204d4350)
4       (ReadLSB w32 0x0
            v0___symfile___input0___0_1_symfile___0))
5    # p->ID == PCM
6
7    (Ult N0:(ReadLSB w32 0x8
        v0___symfile___input0___0_1_symfile___0)
```

```
8        (w32 0x7fffb))
9    # p->NumSamples < MAX_SAMPLES
10
11   (Eq false (Eq (w32 0x0) N0))
12   # p->NumSamples != 0
13
14   (Eq (w32 0x10)
15      (Extract w32 0 (UDiv w64
16                       (ZExt w64
17                         (Shl w32
18                           N1:(ReadLSB w32 0x4
                                    v0___symfile___input0___0_1_symfile___0)
19                            (w32 0x3)))
20                       (ZExt w64 N0))))
21   # p->DataSize*8 / p->NumSamples == 16
22
23   ...
24
25  false []
26  [v0___symfile___input0___0_1_symfile___0])
27 # OK -- Elapsed: 0.02817
28 # Solvable: true
29 #   v0___symfile___input0___0_1_symfile___0 =
30  [80,67,77,32,0,0,1,64,129,120,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,
31 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

ANNEXE C

Captures d'écran et graphiques

C.1 Bzip2

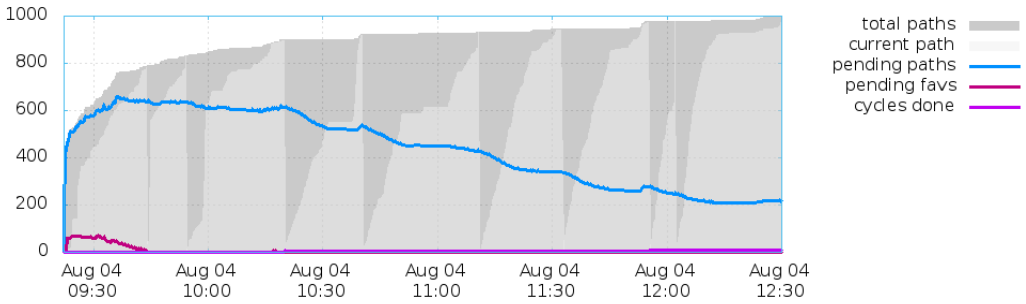


FIGURE C.1: Test de bzip2 par AFL durant 4 heures. Affiche un début de stagnation au niveau des *total paths* dès la première heure.

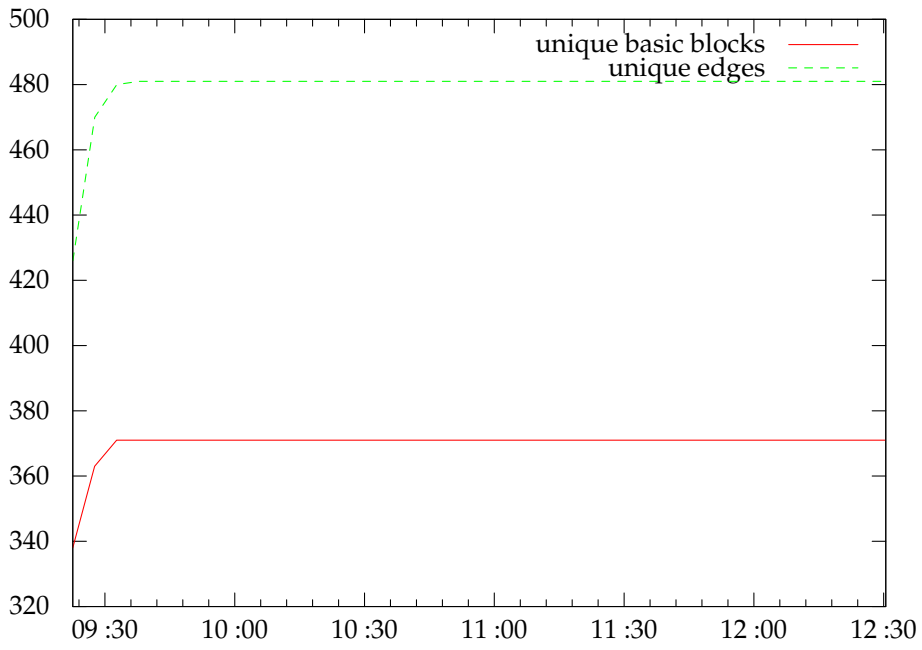


FIGURE C.2: Progression de la couverture de code au cours du test de bzip2 par AFL.

C.2 Parity

```
> ./parity <out/afl/crashes/id:000000,sig:11,src:000018,op:havoc,rep:2
OK
zsh: segmentation fault  ./parity < out/afl/crashes/id:000000,sig:11,src:000018,
op:havoc,rep:2
> ls out/afl/crashes
id:000000,sig:11,src:000018,op:havoc,rep:2
id:000001,sig:11,src:000018,op:havoc,rep:4
id:000002,sig:11,src:000018,op:havoc,rep:8
id:000003,sig:11,src:000018,op:havoc,rep:8
id:000004,sig:11,src:000018,op:havoc,rep:16
id:000005,sig:11,src:000018,op:havoc,rep:4
id:000006,sig:11,src:000018,op:havoc,rep:8
id:000007,sig:11,src:000018,op:havoc,rep:16
id:000008,sig:11,src:000018,op:havoc,rep:8
id:000009,sig:11,src:000018,op:havoc,rep:8
id:000010,sig:11,src:000018,op:havoc,rep:4
id:000011,sig:11,src:000018,op:havoc,rep:32
id:000012,sig:11,src:000018,op:havoc,rep:2
README.txt
> ls out/afl/queue | grep 18
id:000018,sync:s2e,src:000035,+cov
```

FIGURE C.3: Historique du cas de test déclenchant de le bug du programme parity.

Bibliographie

[1] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8) :1978–2001, 2013.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[4] Domagoj Babic. *Exploiting structure for scalable software verification*. PhD thesis, The University Of British Columbia (Vancouver), 2008.

[5] Domagoj Babić and Alan J Hu. Calysto. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 211–220. IEEE, 2008.

[6] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 1–10. IEEE, 2013.

[7] Sébastien Bardin and Philippe Herrmann. Osmose : automatic structural testing of executables. *Software Testing, Verification and Reliability*, 21(1) :29–54, 2011.

[8] Clark Barrett, Leonardo De Moura, and Aaron Stump. Smt-comp : Satisfiability modulo theories competition. In *Computer Aided Verification*, pages 20–23. Springer, 2005.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[10] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints : Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013.

[11] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope : Automatically dissecting malicious binaries. *School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-07-133*, 2007.

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap : a binary analysis platform. In *Computer aided verification*, pages 463–469. Springer, 2011.

[13] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198. ACM, 2011.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[15] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe : automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2) :10, 2006.

[16] Gabriel Campana. Fuzzgrind : un outil de fuzzing automatique. SSTIC, 2009.

[17] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[18] Bing Chen, Qingkai Zeng, and Weiguang Wang. Crashmaker : an improved binary concolic testing tool for vulnerability detection. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1257–1263. ACM, 2014.

[19] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art : Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7) :1758–1773, 2013.

[20] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*. Citeseer, 2009.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform : Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1) :2, 2012.

[22] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the sixth conference on Computer systems*, pages 315–328. ACM, 2011.

[23] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3) :211–217, 2014.

[24] Cormac Flanagan and James B Saxe. Avoiding exponential explosion : Generating compact verification conditions. *ACM SIGPLAN Notices*, 36(3) :193–205, 2001.

[25] Jean-Loup Gailly and Mark Adler. zlib. `http://zlib.net`.

[26] Patrice Godefroid, Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan. Test generation using symbolic execution. In *FSTTCS*, volume 18, pages 24–33. Citeseer, 2012.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[28] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[29] Sidney L Hantler and James C King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3) :331–353, 1976.

[30] Intel. Pin a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[31] IK Isaev and DV Sidorov. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Programming and Computer Software*, 36(4) :225–236, 2010.

[32] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[33] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6) :193–204, 2012.

[34] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[35] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 416–426. IEEE, 2007.

[36] Charlie Miller, Juan Caballero, Noah M Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with bitblaze. *at BlackHat USA*, 2010.

[37] David A Molnar. Dynamic test generation for large binary programs. Technical report, DTIC Document, 2009.

[38] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[39] Brian S Pak. *Hybrid Fuzz Testing : Discovering Software Bugs via Fuzzing and Symbolic Execution*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2012.

[40] Thomas Parsch. Software verification techniques based on floyd's method.

[41] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage : Whitebox fuzzing for security testing. 2012.

[42] Anthony Romano. *Methods for Binary Symbolic Execution*. PhD thesis, Stanford University, 2014.

[43] AMOSSYS SAS. *Conseil & expertise en sécurité*. Mis à jour le 11 mai 2015, AMOSSYS, Consulté le 11 juin 2015. <https://www.amossys.fr/>.

[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[45] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE : a concolic unit testing engine for C*, volume 30. ACM, 2005.

[46] Julian Seward. bzip2. http://bzip.org.

[47] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze : A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[48] Rodney W Topor. *Interactive program verification using virtual programs*. PhD thesis, 1975.

[49] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope : Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[50] Yichen Xie and Alex Aiken. Saturn : A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3) :16, 2007.