En finir avec les faux positifs grâce à l'exécution symbolique robuste

Benjamin Farinier, Sébastien Bardin, Richard Bonichon, Marie-Laure Potet

En finir avec les faux positifs grâce à l'exécution symbolique robuste

Benjamin Farinier[1], Sébastien Bardin[2], Richard Bonichon[2], and Marie-Laure Potet[3]

[1] LRI, Université Paris-Saclay, France
prenom.nom@lri.fr
[2] CEA LIST, Université Paris-Saclay, France
prenom.nom@cea.fr
[3] Univ. Grenoble Alpes, Verimag, France
prenom.nom@univ-grenoble-alpes.fr

Résumé

L'exécution symbolique est une technique de vérification formelle par sous-approximation ayant fait ses preuves en recherche de bogues, notamment grâce à son absence de faux positifs : un bogue trouvé est un bogue réel. Cependant, si cette propriété est vraie dans le cas où l'utilisateur contrôle toutes les entrées du programme, les choses se compliquent quand certaines entrées sont hors de son contrôle, typiquement l'environnement. L'exécution symbolique devient alors *fragile* dans le sens où elle peut produire des faux positifs. Ce cas se rencontre particulièrement en recherche de vulnérabilités, où les failles cherchées doivent être reproductibles. Dans cet article nous montrons comment l'utilisation de quantificateurs permet de passer du problème de l'accessibilité à celui de l'*accessibilité robuste* et proposons une modélisation cohérente en tant que sous-approximation. Ces quantificateurs sont ensuite éliminés et les formules obtenues simplifiées afin de limiter au maximum l'impact sur le temps de résolution. Il en résulte ainsi une analyse par exécution symbolique efficace et *robuste* vis-à-vis de son environnement, réellement exempte de faux positifs.

1 Introduction

Contexte. La vérification de programmes est un succès indéniable pour les logiciels critiques. De nombreux efforts poussent depuis plusieurs années à l'adapter à des logiciels non critiques. Notamment, *l'exécution symbolique* [4] vise à explorer une partie des chemins d'un programme afin de trouver des bogues. Chaque nouveau chemin est accompagné d'une entrée obtenue par raisonnement symbolique et Satisfiabilité Modulo Théorie (SMT) [2] permettant de l'activer. Cette méthode rencontre un énorme succès depuis quelques années [10] pour sa capacité à passer sur des codes complexes et son *absence de faux positifs* : un bogue rapporté est un bogue certain. L'exécution symbolique fait ainsi partie des techniques de vérification par *sous-approximation*.

Problème. Cependant, en pratique *les faux positifs existent* [3, 5]. Ils proviennent d'abstractions de certains morceaux de code absents ou trop compliqués à gérer, par exemple lors d'une analyse niveau C d'un code contenant de l'assembleur, d'une modélisation de l'état initial ou de l'environnement imparfaite, par exemple en analyse de code binaire. Ce problème est connu dans la communauté de l'exécution symbolique, mais pas véritablement étudié ni quantifié. Par ailleurs, il est plutôt vu comme une fatalité, pas forcément gênante si la méthode trouve suffisamment de bogues par ailleurs. C'est cependant un problème sérieux pour l'utilisateur, qui ne peut plus faire confiance aux taux de couverture calculés (test) ou doit passer du temps à analyser un bogue retourné non confirmé par l'entrée associée (sécurité). Le problème est

particulièrement saillant au niveau de l'analyse de code binaire, car les entrées initiales et les interactions avec l'environnement ne sont pas aisées à établir a priori.

En pratique, des solutions ad hoc peuvent diminuer le nombre de faux positifs, mais elles n'en garantissent pas l'absence et parfois en introduisent de nouveaux ou surchargent complètement le solveur automatique.

Proposition. Nous visons à développer une *exécution symbolique robuste*, c'est-à-dire réellement sans aucun faux positif. Pour cela nous proposons un nouveau cadre, *l'accessibilité robuste*, dans lequel les entrées sont partitionnées en *contrôlées* (par l'utilisateur) et *non contrôlées* (environnement). Une solution est dite robuste si elle atteint le but voulu indépendamment des valeurs non contrôlées. Ce cadre permet d'encoder les cas gênants évoqués plus haut. Cet article présente nos premiers résultats sur l'exécution symbolique robuste.

— Nous définissons formellement le cadre d'accessibilité robuste et montrons intuitivement son intérêt pratique ;

— Nous proposons une révision de l'exécution symbolique, dite exécution symbolique robuste, destinée à résoudre le problème de l'accessibilité robuste et à éliminer les faux positifs ;

— Finalement, nous décrivons une première implémentation dans l'outil d'analyse de code binaire Binsec [6, 7] et rapportons de premiers résultats expérimentaux encourageants.

2 Avant-propos sur l'exécution symbolique

L'exécution symbolique [4] est une approche formelle au test automatisé de logiciel, fondée sur la sémantique du code du programme. Cette approche repose sur l'idée suivante : étant donné un chemin dans un programme, il est possible de calculer une formule logique sur les *entrées du programme*[1], appelée *prédicat de chemin*, dont une solution est une entrée de test exerçant ledit chemin. Bien que cela soit souhaitable, il n'est pas nécessaire que toutes les entrées exerçant le chemin soient solutions du prédicat puisque la technique est par définition une sous-approximation — seule une partie des chemins est explorée.

```
int main () {
  int x = input();
  int y = input();
  int z = 2 * y;
  if (z == x) {
    if (x > y + 10)
      printf("Success!\n");
  }
  printf("Failure...\n");
}
```

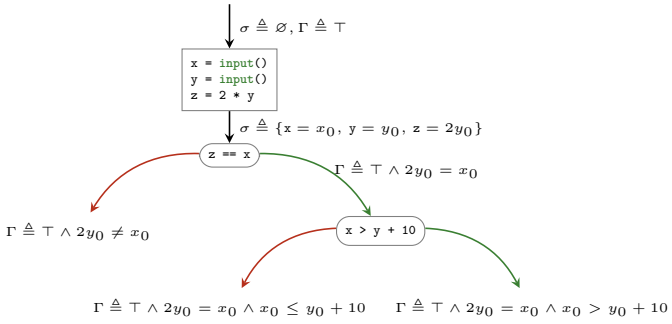

Figure 1 – Exécution symbolique sur un petit programme

Sur l'exemple en Fig. 1, l'exécution symbolique va énumérer les trois chemins possibles de ce petit programme. Chacun d'entre eux correspond à un prédicat de chemin Γ différent. Celui-ci est construit par la conjonction des contraintes générées aux points de branchement du

1. Au sens large : ligne de commande, arguments, fichiers, environnement, etc.

programme, en fonction des valeurs logiques des variables dans l'environnement σ. La faisabilité du chemin est ici équivalente à la satisfiabilité de la formule qui le représente.

Nous utilisons ensuite un solveur automatique, typiquement un solveur SMT. Lorsque ces solveurs répondent positivement à la question de la satisfiabilité, les modèles de la formule générée font voir des jeux d'entrées exerçant le chemin exécuté symboliquement. Ainsi, le modèle $\{x_0 = 0,\ y_0 = 1\}$ est une solution (parmi d'autres) au chemin dont la contrainte est $\top \wedge 2y_0 \neq x_0$. Notons que les formules générées sont naturellement *non quantifiées* et tombent dans le *fragment décidables* des théories employées — ici vecteurs de bits (variables) et tableaux (mémoire) — et que c'est la *génération de modèles* plutôt que la simple question de la satisfiabilité qui importe.

Une exécution symbolique est dite *complète* si elle couvre toutes les exécutions concrètes possibles. Une exécution symbolique est dite *correcte* si chaque solution générée pour chaque chemin parcouru est *correcte*, c'est-à-dire qu'elle suivra le chemin prévu lors de son exécution concrète. Cette définition esquisse en pointillés la notion de *faux positifs*, des solutions venant d'un prédicat de chemin incorrect et ne suivant pas le chemin prévu lors de l'exécution concrète.

3 Motivation

```c
#define SIZE

void get_secret (char secr[]) {
// Retrieve the secret
}

void read_input (char src[], char dst[]) {
  int i = 0;
  while (src[i]) {
    dst[i] = src[i];
    i++;
  }
}

int validate (char secr[], char inpt[]) {
  int b = 1;
  for (int i = 0; i < SIZE; i++) {
    b &= secr[i] == inpt[i];
  }
  return b;
}

int main (int argc, char *argv[]) {
  char secr[SIZE];
  char inpt[SIZE];

  if (argc != 2) return 0;

  get_secret(secr);
  read_input(argv[1],inpt);

  if (validate(secr, inpt)) {
    printf("Success!\n");
  }
  else {
    printf("Failure...\n");
  }
}
```

FIGURE 2 – Programme pour lequel nous cherchons une entrée permettant d'atteindre « Success ! »

Prenons en guise d'exemple le programme donné en Fig. 2 comportant, en plus de la fonction `main` les fonctions `get_secret`, `read_input` et `validate`. Schématiquement, la fonction `get_secret`, dont le code n'est pas donné, récupère un secret à deviner, inconnu de l'utilisateur et imprévisible (fonction cryptographique, etc.), et l'écrit dans le tampon prévu à cet effet. La fonction `read_input` copie l'entrée utilisateur dans le tampon qui lui est consacré tandis que la fonction `validate` vérifie que l'entrée utilisateur est égale au secret à deviner. La fonction `main`, après avoir contrôlé que le bon nombre d'arguments a été donné, utilise ces fonctions pour vérifier si l'entrée correspond au secret, et affiche selon le cas « Success ! » ou « Failure... ».

Objectif. Notre but est de trouver une entrée permettant d'atteindre le branche « Success ! ». Nous montrons tout d'abord comment une utilisation directe de l'*exécution symbolique classique*

au niveau du code C conduit à une première solution non robuste, c'est-à-dire un *faux positif*. Nous montrons aussi comment notre méthode d'*exécution symbolique robuste* évite cet écueil (pas de faux positif), sans toutefois trouver de « vraie » solution au problème. Nous considérons ensuite la sémantique au niveau binaire du code associé. L'exécution symbolique classique conduit toujours à un faux positif, mais cette fois l'*exécution symbolique robuste* trouve bien une « vraie » solution, en passant par un dépassement de tampon.

Premier essai : sémantique haut niveau. Essayons donc par exécution symbolique classique de trouver une entrée qui nous permette d'atteindre la branche « Success ! ». Puisque son contenu provient d'une entrée utilisateur, le tableau de caractères `inpt` va être modélisé par une première variable symbolique i. De même le tableau de caractères `secr` va être modélisé par une seconde variable symbolique s, car son contenu provient de l'exécution d'un code ne faisant pas partie de l'analyse (appel à une bibliothèque cryptographique externe, etc). Finalement, la contrainte à satisfaire pour valider le problème va être $i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$, où $i_{[0,\,\texttt{SIZE}-1]}$ (resp. $s_{[0,\,\texttt{SIZE}-1]}$) dénote les valeurs de i (resp. de s) aux indices allant de 0 à $\texttt{SIZE}-1$. Mais il s'avère que cette contrainte a pour solution triviale $\left\{i_{[0,\,\texttt{SIZE}-1]} = 0,\ s_{[0,\,\texttt{SIZE}-1]} = 0\right\}$, qui ne sera que très improbablement une solution réelle à notre problème.

Ce *faux-positif* vient du fait que les outils d'exécution symbolique considèrent toutes les entrées du programme de la même manière, c'est-à-dire *contrôlées par l'utilisateur*, ce qui donne ici le prédicat de chemin $\exists i.\exists s.i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$ — nous ajoutons les quantifications existentielles sur i et s pour bien faire ressortir les entrées ; en pratique ce sont juste des variables libres puisque nous effectuons une requête de satisfiabilité. Or ici, puisque s est hors du contrôle de l'utilisateur, une modélisation appropriée du problème est $\exists i.\forall s.i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$, qui nécessite des quantificateurs universels. Cette contrainte n'est cette fois-ci pas satisfiable, mais, à défaut de solutions, il n'y a plus de faux positifs.

Second essai : sémantique bas niveau. Si l'on souhaite trouver une solution à notre problème, il nous faut aller plus en détail dans le fonctionnement de l'exécution symbolique et dans la modélisation de notre programme. On remarque premièrement que la fonction `read_input` lit l'entrée utilisateur jusqu'à rencontrer le caractère nul de fin de chaîne. Du côté de l'exécution symbolique, cela se matérialise par l'introduction d'une constante N implicitement définie par le nombre d'itérations de la boucle de lecture et représentant la longueur de l'entrée utilisateur. Deuxièmement, remarquons que les variables locales `inpt` et `secr` sont des tableaux de caractères de taille fixe qui sont alloués directement dans la pile, consécutivement. Du côté de la modélisation, cela se matérialise par l'introduction de deux variables symboliques supplémentaires, m_0 représentant l'état mémoire du programme à un instant donné, et p_0 un pointeur vers la pile du programme. Les variables s, m_0 et p_0 sont universellement quantifiées car l'utilisateur ne peut directement fixer leur valeur de sorte à résoudre le problème. En notant $\triangleq$ la définition de variables intermédiaires, la contrainte que l'on cherche à satisfaire devient :

$$
\begin{aligned}
&\exists i.\,\forall s.\,\forall m_0.\,\forall p_0.\\
&\quad p_1 \triangleq p_0 - \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0\,[p_1,\, p_1 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_2 \triangleq m_1\,[p_2,\, p_2 + N - 1] \leftarrow i\\
&m_2\,[p_1,\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2,\, p_2 + \texttt{SIZE} - 1]
\end{aligned}
$$

Encore une fois, cette contrainte comporte des quantificateurs, en particulier universels, qui augmentent significativement la difficulté à trouver une solution pour les solveurs. On

cherche donc à les éliminer en utilisant une approche par teinte [8]. Sur notre exemple, on peut ainsi remplacer les quantifications universelles par des existentielles sur s et m_0 si on ajoute la contrainte $[p_1,\, p_1 + \texttt{SIZE} - 1] \subseteq [p_2,\, p_2 + N - 1]$ équivalente à $N \geq 2 \cdot \texttt{SIZE}$, soit donc :

$$
\begin{aligned}
&\exists i.\, \exists s.\, \exists m_0.\, \forall p_0.\\
&\quad p_1 \triangleq p_0 - \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0\,[p_1,\, p_1 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_2 \triangleq m_1\,[p_2,\, p_2 + N - 1] \leftarrow i\\
&m_2\,[p_1,\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2,\, p_2 + \texttt{SIZE} - 1]\\
&\wedge\ N \geq 2 \cdot \texttt{SIZE}
\end{aligned}
$$

On simplifie ensuite cette contrainte en réduisant les *read-over-write* (lectures suivant une écriture) [9]. On obtient la contrainte $\exists i.\ i_{[0,\,\texttt{SIZE}-1]} = i_{[\texttt{SIZE},\,2\cdot\texttt{SIZE}-1]} \wedge N \geq 2 \cdot \texttt{SIZE}$ ne faisant plus référence qu'à la variable symbolique i existentiellement quantifiée. Ce dernier quantificateur pouvant simplement être enlevé (requête de satisfiabilité), nous devons donc trouver un modèle pour la formule désormais sans quantificateur $i_{[0,\,\texttt{SIZE}-1]} = i_{[\texttt{SIZE},\,2\cdot\texttt{SIZE}-1]} \wedge N \geq 2 \cdot \texttt{SIZE}$.

Pour par exemple $\texttt{SIZE} = 8$, une solution possible à cette formule est `abcdefghabcdefgh`, qui permet effectivement d'attendre la branche « Success ! » par dépassement de tampon dans la fonction `read_input`.

4 Exécution symbolique robuste

Accessibilité robuste. Nous rappelons tout d'abord la notion classique d'accessibilité. Pour un programme P sur des entrées (a, x), nous disons qu'un objectif (l, φ) — où l est une localisation dans le code et φ un prédicat local sur l'état du programme à la localisation l, est *accessible* s'il existe un couple d'entrées (a_0, x_0) tel que l'exécution de P sur (a_0, x_0) atteint effectivement la localisation l dans un état qui satisfait φ. Nous noterons $P(a_0, x_0) \rightsquigarrow (l, \varphi)$.

Nous définissons maintenant la notion d'*accessibilité robuste*. Pour un programme P sur des entrées (a, x) où a est contrôlée et x ne l'est pas, nous disons qu'un objectif (l, φ) est *accessible de manière robuste* s'il existe une entrée a_0 telle que pour toute entrée x_0 on ait $P(a_0, x_0) \rightsquigarrow (l, \varphi)$. Une telle donnée d'entrée utilisateur a_0 permet donc d'atteindre l'objectif quelle que soit la donnée de l'environnement.

Exécution symbolique robuste. Nous adaptons maintenant le principe de l'exécution symbolique au contexte de l'accessibilité robuste.

Un *prédicat de chemin robuste* doit désormais assurer l'accessibilité robuste, et non plus seulement l'accessibilité, du chemin visé. Il est en définitive aisé de calculer un tel prédicat robuste : il suffit d'utiliser le calcul symbolique habituel, puis de quantifier universellement les variables non contrôlées. On obtient ainsi un prédicat quantifié de la forme $\exists a.\forall x.\Gamma(a, x)$. Cette étape demande que l'utilisateur ait spécifié au préalable les entrées contrôlées et non contrôlées. Nous discutons ce point dans le prochain paragraphe.

Le problème est ensuite de pouvoir résoudre de tels prédicats quantifiés, car l'ajout de quantificateurs rend de nombreuses théories indécidables — c'est le cas par exemple de la théorie des tableaux. Heureusement, nous pouvons réutiliser au choix : certaines extensions récentes des solveurs SMT (par exemple, quantificateurs sur les vecteurs de bits), ou notre méthode d'élimination des quantificateurs de manière à maintenir une génération de modèles correcte [8].

Spécification des entrées utilisateurs. Par rapport à l'exécution symbolique classique, notre méthode demande donc à l'utilisateur de spécifier quelles entrées sont contrôlées ou non. Cet aspect est crucial : si l'utilisateur déclare comme contrôlées des entrées qui ne le sont pas dans le système réel, alors notre méthode retournera des faux positifs — pas par rapport au problème sous-jacent d'accessibilité robuste, mais par rapport au système réel.

Nous pouvons par contre remarquer que si l'utilisateur a déclaré « trop » d'entrées non contrôlées, alors notre technique ne peut pas produire de faux positifs — elle perdra juste en généricité. Une manière *robuste* de faire est donc de considérer que par défaut une entrée n'est contrôlée que si l'utilisateur l'a spécifié explicitement, là où l'exécution symbolique classique correspond exactement au choix inverse.

Ainsi dans notre implantation, en mode robuste, les entrées utilisateurs doivent être explicitement marquées comme contrôlées pour être modélisées par des variables existentielles. À titre d'exemple, le fichier de spécification pour l'analyse du programme en Fig. 2 est donné en Fig. 3.

```
controlled argc<32>;
@[esp<32> + 4<32>, 4] := argc<32>;

argv0_ptr := esp<32> + 0x00000100;              controlled argv0<160>;
argv1_ptr := esp<32> + 0x00000200;              controlled argv1<160>;

@[esp<32> +  8<32>, 4] := argv0_ptr;            @[argv0_ptr, 20] := argv0<160>;
@[esp<32> + 12<32>, 4] := argv1_ptr;            @[argv1_ptr, 20] := argv1<160>;
```

FIGURE 3 – Spécification des entrées contrôlées par l'utilisateur

Implémentation. Les différentes étapes nécessaires à l'exécution symbolique robuste sont implémentées pour la théorie des vecteurs de bits et tableaux dans TFML, le module de traitement des formules logiques de l'outil d'exécution symbolique BINSEC [6]. Côté élimination, nous utilisons la procédure générique d'élimination des quantificateurs par inférence de conditions d'indépendance présentée dans [8], en particulier son traitement de la théorie des tableaux : pour chaque tableau dans la formule un tableau fantôme est introduit permettant de suivre quelles cellules du tableau initial dépendent de variables universellement quantifiées. Cela évite l'introduction d'un grand nombre de if-then-else et permet de faire bénéficier aux contraintes introduites de nos simplifications sur les tableaux. Côté simplification donc, les règles de réécritures présentées dans [9] sont systématiquement appliquées à la construction des termes. De plus le traitement des *read-over-write* (théorie des tableaux) est appliqué une fois avant l'élimination des quantificateurs afin d'en faciliter les raisonnements sur les pointeurs, et une fois après pour réduire les contraintes introduites.

5 Évaluation expérimentale

On évalue l'impact de notre approche selon deux critères : 1) le nombre de vrais positifs, qui doit être dégradé le moins possible ; et 2) le nombre de faux positifs, qui doit être réduit à zéro. Pour ce faire, on considère un ensemble de challenges de type `crackme` dont le but est de trouver une entrée amenant le programme à dévoiler son secret. L'intérêt de ces programmes est qu'ils permettent de valider simplement une solution en rejouant le challenge avec. On compare notre exécution symbolique robuste avec élimination des quantificateurs et simplifications à

l'exécution symbolique classique avec simplification, et à une exécution symbolique robuste avec simplification mais sans élimination des quantificateurs.

Les expérimentations sont réalisées sur un processeur Intel Core i7-4712HQ @ 2.30GHz avec 16Go de RAM, et les résultats sont présentés dans la Table 1. L'approche classique trouve jusqu'à 12 solutions correctes pour ces challenges en fonction du solveur sous-jacent. Cependant, elle produit aussi de 9 à 12 faux positifs ne permettant pas de résoudre les challenges correspondant. L'approche robuste sans élimination des quantificateurs est, elle, bien exempte de faux positifs. Néanmoins le nombre de vrais positifs est fortement impacté, tombant à 7 pour le meilleur solveur. Enfin notre approche robuste avec élimination des quantificateurs permet de retrouver tous les vrais positifs de l'approche classique tout en restant exempt de faux positifs.

TABLE 1 – Comparaison du nombre de vrais et de faux positifs pour l'exécution symbolique classique, robuste sans élimination et robuste avec élimination des quantificateurs

	approche classique			approche robuste			approche robuste + élim.		
	SAT correct	SAT incorrect	UNSAT ou UNKNOWN	SAT correct	SAT incorrect	UNSAT ou UNKNOWN	SAT correct	SAT incorrect	UNSAT ou UNKNOWN
Boolector	**12**	11	1	N/A	**0**	24	**12**	**0**	12
CVC4	**7**	9	8	5	**0**	19	**7**	**0**	17
Yices	**7**	11	6	N/A	**0**	24	**7**	**0**	17
Z3	**12**	12	0	7	**0**	17	**12**	**0**	12

Application à l'analyse de vulnérabilités. Précisons tout d'abord que notre exécution symbolique robuste permet de trouver une solution correcte à l'exemple donné en Fig. 2. Nous l'avons aussi appliquée avec succès à des cas d'études réels issus de l'analyse de vulnérabilité, comme par exemple une version adaptée d'une faille de sécurité nommée « Back to 28 » [1]. Dans sa version originale, elle permettait en appuyant successivement 28 fois sur la touche retour arrière de contourner la procédure d'authentification du chargeur d'amorçage GRUB2. Dans la version de notre cas d'étude, le code à l'origine de cette faille est repris à l'identique à ceci près que la corruption d'adresse de retour est remplacée par une corruption des données en mémoire similaire à notre exemple. Ici encore, l'exécution symbolique classique retourne une solution fragile, tandis que les solveurs sous-jacents échouent à résoudre les formules produites par l'exécution symbolique robuste sans élimination des quantificateurs. Notre exécution symbolique robuste trouve une solution en 80 secondes avec élimination des quantificateurs, et même en 30 secondes en rajoutant les simplifications de tableau.

6 Autres approches, conclusion et travaux futurs

Autres approches. Comme dit auparavant, il n'existe pas d'approche systématique et correcte au problème de l'accessibilité robuste. Les outils d'exécution symbolique actuels se contentent de résoudre le problème de l'accessibilité classique (non robuste), et en pratique les utilisateurs essaient de pallier le problème des faux positifs de manière ad hoc.

Par exemple, pour le problème de l'état initial mal défini, particulièrement criant pour de l'analyse niveau binaire, on rencontre les mitigations suivantes :

— Initialisation de la mémoire à une valeur arbitraire, par exemple 0 : cela enlève certains faux positifs mais peut en ajouter d'autres, dans le cas où 0 établit la solution du prédicat de chemin sans pour autant correspondre à une réalité de l'OS / architecture sous-jacente.

— Initialisation de la mémoire à celle observée à l'exécution (concrétisation) : là encore on enlève certains faux positifs (une partie de ces valeurs est effectivement constante d'une exécution à l'autre), mais pas tous (certaines valeurs ne sont pas constantes d'une exécution à l'autre) ; par ailleurs la contrainte induite peut devenir trop imposante pour être gérée par les solveurs.

Conclusion et travaux futurs. Le cadre de l'exécution symbolique robuste permet de remédier à ce problème de manière définitive, à condition que : 1) les solveurs arrivent à gérer les formules quantifiées générées ; et 2) l'utilisateur ait bien spécifié les entrées contrôlées et non contrôlées. Nos travaux récents [8, 9] montrent comment une approche par pré-traitement permet de réutiliser les solveurs (*quantifier-free*) pour résoudre le point (1), et nous avons implémenté une preuve de concept dans l'outil Binsec.

Le point (2) est évidemment un problème puisqu'il repose sur l'utilisateur. Comme expliqué précédemment, notre technique ne peut pas produire de faux positifs quand « trop » d'entrées sont déclarées non contrôlées, et c'est pour cela que par défaut, nous considérons toutes les entrées non contrôlées. La tâche fastidieuse de spécifier explicitement quelles entrées sont contrôlées revient alors à l'utilisateur. Nos prochains travaux s'orientent justement vers l'aide à l'identification des entrées contrôlées.

Références

[1] http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html.

[2] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[4] C. Cadar and K. Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E : a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278, 2011.

[6] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *SANER, Suita, Osaka, Japan, March 14-18, 2016*, pages 653–656, 2016.

[7] A. Djoudi and S. Bardin. BINSEC : Binary Code Analysis with Low-Level Regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 212–217, 2015.

[8] B. Farinier, S. Bardin, R. Bonichon, and M.-L. Potet. Model generation for quantified formulas : A taint-based approach. In *30th International Conference on Computer Aided Verification (CAV 2018)*, 2018.

[9] B. Farinier, R. David, S. Bardin, and M. Lemerre. Arrays made simpler : An efficient, scalable and thorough preprocessing. In *22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2018)*, 2018.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. In *PLDI, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.