Triton : Framework d'exécution concolique et d'analyses en *runtime*

Jonathan Salwan ⋆ et Florent Saudel ⋆⋆
jsalwan@quarkslab.com
florent.saudel@etu.u-bordeaux.fr

Quarkslab, Université de Bordeaux

Résumé Triton est un framework d'exécution concolique basé sur Pin. L'objectif de Triton est d'apporter des composants supplémentaires au framework Pin déjà existant afin de faciliter la mise en place d'analyses concoliques poussées. Triton s'utilise depuis des *bindings* Python et propose des composants tel que de l'analyse par teinte (*taint analysis*), un moteur d'exécution symbolique, un moteur de prise d'instantané (*snapshot*), une sémantique des instructions x86-64 en SMT2 ainsi qu'une interface avec le SMT *solver* Z3. Basé sur ces composants, il est possible de développer des outils externes en Python permettant d'effectuer du *fuzzing* symbolique, d'analyser des traces, d'effectuer des analyses en *runtime* pour de la recherche de vulnérabilités, etc.

1 Introduction

Les binaires évoluent de plus en plus vite, que cela soit en terme de taille comme en terme de complexité. L'utilisation de techniques d'obfuscation pour protéger des binaires tend à se généraliser, compliquant et ralentissant la tâche de l'analyste. Les outils d'analyse se doivent donc également d'évoluer pour faire en sorte de ne pas être noyés dans la quantité d'informations à analyser.

Il existe plusieurs techniques d'analyse que cela soit en statique ou en dynamique. Il n'y a pas de mauvaise ou bonne méthode, chacune d'entre elles a ses avantages et ses inconvénients.

Les analyses statiques permettent de couvrir un maximum de code mais ne permettent pas d'avoir les valeurs concrètes des variables à un instant T. L'idée est de définir au plus près l'ensemble des valeurs possibles

⋆. Ce projet a été fait dans le cadre de notre master CSI à Bordeaux sous la supervision d'Emmanuel Fleury. Je remercie donc Quarkslab pour le temps libre qui m'a été donné.

⋆⋆. Je remercie Georges Bossert pour sa relecture.

des registres et de la mémoire, en un point donné du programme. Un type d'analyse statique connue est l'interprétation abstraite. Son rôle est de surévaluer l'ensemble des valeurs et de vérifier certaines propriétés en conséquence. Même si cette sur-approximation peut engendrer des faux positifs, cette approche n'est pas délaissée par le milieu de la sécurité informatique. Par exemple l'interprétation abstraite *Value-set Analysis* [2] permet de de compléter un graphe de flot de contrôle (CFG) en lui rajoutant des arrêtes issues de sauts indirects. Pour cela, elle calcul un sur-ensemble des valeurs des pointeurs du binaire.

Les analyses dynamiques quant à elles permettent de connaître l'état de la mémoire et des registres en chaque point du programme mais n'offrent pas de couverture de code. Cela signifie que l'analyse se fait sur une unique trace. En dynamique, la principale difficulté est de pouvoir parcourir l'ensemble de l'arbre d'exécution [1] [11]. Triton dispose d'un moteur de prise d'instantanée (*snapshot*) afin de remonter l'exécution au sein de cet arbre. L'exécution symbolique lui permet de générer les entrées nécessaires afin de suivre un autre chemin et ainsi couvrir une autre portion de code. Il existe plusieurs travaux sur ce sujet comme l'outil interne de Microsoft appelé *SAGE* [14] et les travaux présenté par Gabriel Campana [3] au SSTIC 2009 et Sébastien Lecomte [12] au SSTIC 20014.

Le projet Triton que nous détaillons dans cet article, rentre dans la catégorie des analyses dynamiques. Il est modélisé en tant que surcouche au framework Pin [9]. Il propose des composants supplémentaires comme un moteur de teint, un moteur symbolique, une gestion et résolution des contraintes, un système de rejeu par instantanée et une conversion des instructions en représentation SMT2-LIB.

2 Exécution symbolique

Avant de rentrer dans la description du projet Triton, ce chapitre effectue un rappel sur l'exécution symbolique.

Dans le domaine de l'exécution symbolique, il existe deux grands axes qui sont l'exécution symbolique statique (*SSE* pour *Static Symbolic Execution*) et l'exécution symbolique dynamique (*DSE* ou aussi appelée exécution concolique). Le choix entre le SSE et le DSE va se faire en fonction du type d'analyse que vous effectuez et des informations à disposition.

1. Cet arbre comprend tous les chemins possibles d'un programme (potentiellement infini).

Le SSE est principalement utilisé pour vérifier des propriétés sur un code donné alors que le DSE est utilisé pour la construction des contraintes correspondant au flux de contrôle d'une exécution.

Étant donné que le projet Triton travaille sur de l'exécution dynamique, nous détaillerons plus en détail l'approche dynamique.

Mais auparavant, nous rappellons le principe même de l'exécution symbolique.

2.1 Principe et fonctionnement de l'exécution symbolique

Le rôle d'une exécution symbolique est de produire une formule logique correspondant au programme ou à un chemin du programme.

De façon analogue au calcul formel, celle-ci se base sur des variables symboliques (ou symboles). Ici, le mot variable prend le sens mathématique. Elle représente une inconnue dans la formule. Autrement dit, c'est une variable « libre » et peut donc être substituée par n'importe quelle valeur.

Comme exemple, prenons le code simpliste écrit en langage C exposé dans le Listing 1.

```
1 int f(int x, int y) {
2     int val = x*y + 1
3
4     if (val <= x*y)
5         crash(); //erreur
6
7     return val;
8 }
```

Listing 1. Exemple de code C

Si nous réalisons l'appel suivant : $f(1, 3)$, il est facile de voir que le résultat obtenu est 4 pour une exécution normale (concrète). La Table 1 présente le déroulement de l'exécution symbolique en parallèle de celle réelle. L'exécution s'effectue sur les deux contextes : concret et symbolique. À chaque variable du programme est associée une valeur concrète (réelle) et une valeur symbolique. Ces associations sont représentées à l'aide de deux ensembles de fonctions. Un allant de l'ensemble des variables $\{x, y\}$ vers l'ensemble concret $\mathbb{Z}$ et un autre partant du même ensemble des variables mais allant vers l'ensemble symbolique $\{x_s, y_s, x_s * y_s, \ldots\}$.

Au départ, les arguments x et y de la fonction f peuvent prendre n'importe quelle valeur, par conséquent nous leur affectons les valeurs symboliques x_s et y_s (première ligne du Table 1).

À partir de là, chaque instruction du programme met à jour le contexte symbolique selon une règle précise. Chaque instruction concrète possède une traduction symbolique équivalente. C'est-à-dire qu'il existe une règle (ou une composition de règles) définissant son action (son *sens*) sur le contexte symbolique. La représentation des instructions symboliques se fait à l'aide d'un langage appelé Langage Intermédiaire (IL en anglais) ou encore Représentation Intermédiaire (IR en anglais). Un exemple complet d'un tel IL est donné dans [15].

Pour réaliser l'exécution symbolique de cet exemple 1, il suffit de définir l'affectation, l'addition, la multiplication et les sauts conditionnels. De façon informelle, l'affectation met à jour la valeur symbolique associée à une variable du programme (ligne 2). Les opérations arithmétiques correspondent à celles mathématiques que nous connaissons tous (ligne 2). Par exemple, l'addition de trois variables a, b et c correspond à l'expression $a + b + c$. Les sauts conditionnels font apparaître les contraintes influant sur le flux d'exécution. Si la condition du *if* est remplie, elle devient la contrainte pour le chemin que nous parcourons. A l'inverse, si elle n'est pas remplie, notre contrainte est sa négation (ligne 4).

Ligne	Contexte concret	Contexte symbolique	Contrainte
1	$\{x \to 1, y \to 3\}$	$\{x \to x_s, y \to y_s\}$	$\emptyset$
2	$\{x \to 1, y \to 3, val \to 4\}$	$\{x \to x_s, y \to y_s, val \to x_s * y_s + 1\}$	$\emptyset$
4	$\{x \to 1, y \to 3, val \to 4\}$	$\{x \to x_s, y \to y_s, val \to x_s * y_s + 1\}$	$\{\neg(val \leq x_s * y_s)\}$
7	$\{x \to 1, y \to 3, val \to 4\}$	$\{x \to x_s, y \to y_s, val \to x_s * y_s + 1\}$	$\{x_s * y_s + 1 > x_s * y_s\}$

Table 1. Exécution concolique de $f(1, 3)$.

2.2 Précision sur la notion de sémantique

La traduction du concret vers symbolique doit faire apparaître la sémantique des instructions. L'effet d'une instruction symbolique sur son contexte doit être équivalent à celui de l'instruction de code sur le contexte concret. Rappelons que nous souhaitons travailler sur un code assembleur.

L'instruction *add* permet de réaliser une addition, mais peut-on utiliser la traduction arithmétique simple que nous venons de voir ? La réponse est non car cette instruction n'agit pas seulement sur le registre de destination mais aussi sur le registre EFLAGS. Comme ces registres influent de manière indirecte sur le flux d'exécution, l'IR que nous devons construire doit aussi prendre en compte leur modifications. Voilà pourquoi l'IR doit représenter la sémantique d'un programme et pas seulement sa syntaxe.

Afin que notre exécution soit valide, il est nécessaire de travailler dans la logique adéquate. Comme le montre l'exemple du Listing 1, la condition du *if* est équivalent à l'inéquation $x * y + 1 < x * y$. Si x et y appartiennent $\mathbb{Z}$ alors aucun couple (x, y) ne peut la satisfaire. Par conséquent, l'instruction *crash* semble impossible à atteindre. Pourtant, ce programme lève une erreur sur l'entrée (`0x1, 0x7fffffff`) à cause d'un *integer overflow*. Jusqu'à présent, nous réfléchissions dans la logique arithmétique classique. Cependant la logique des ordinateurs est celle des *Bit Vectors* où les entiers sont codés sur un nombre de bits fixe. Il est donc important d'adopter cette logique au sein de notre exécution symbolique et lors de la résolution des contraintes.

Nous venons de voir le fonctionnement général d'une exécution symbolique sur un cas simple et ce pour une seule exécution. Nous allons voir comment une exécution symbolique s'effectue sur tout un programme.

2.3 Parcours de l'arbre d'exécution

L'un des principaux inconvénients d'une analyse dynamique est la couverture de code. Lors d'une passe dynamique, nous ne pouvons analyser qu'une seule trace, en d'autres mots, nous analysons une séquence d'instructions empruntant un chemin satisfaisant une suite de conditions (appelé PC pour *Path Condition*).

Une fonction peut également contenir plusieurs branchements dus à des conditions. Si nous prenons l'exemple de la Figure 1, cette trace contient 3 conditions qui ont été satisfaites ($\pi_1 \wedge \pi_2 \wedge \pi_3$).

Le fait d'effectuer une exécution symbolique nous permet de connaître à chaque point du programme (ici nos π_x) quelle est la formule vérifiée pour prendre le branchement Vrai (V) ou Faux (F). Il nous est donc possible de prendre le branchement V en satisfaisant sa formule (π_x) ou prendre le branchement F en satisfaisant sa négation ($\neg\pi_x$).

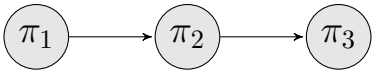
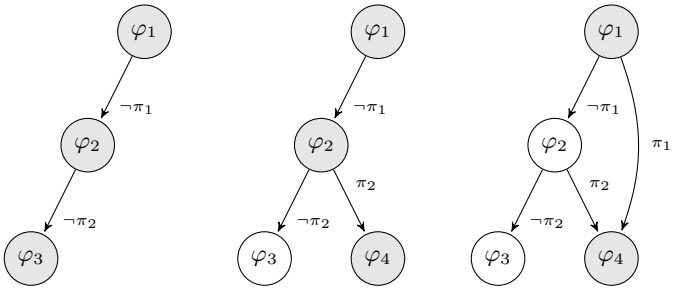


Figure 1. Une trace avec 3 conditions

Pour effectuer une couverture de code en utilisant une approche de type DSE, il nous faut rejouer les traces ou chemins en modifiant les entrées concrètes pour emprunter à chaque itération un branchement différent.

Cette approche est schématisée par la Figure 2. L'idée est d'effectuer une première itération avec des valeurs concrètes aléatoires, ce qui nous permet de construire la première formule symbolique qui satisfait le chemin $\neg\pi_1 \wedge \neg\pi_2$. Basé sur la formule construite lors de la précédente itération, nous rejouons la trace et satisfaisons le dernier branchement $\neg\pi_1 \wedge \pi_2$, puis une troisième itération qui satisfait π_1. Pour résumer, l'opération est répétée en effectuant du *backtracking* sur tous les branchements qui se trouve sur nos traces jusqu'à ne plus avoir de branchement à prendre.



(a) Itération 1 (b) Itération 2 (c) Itération 3

Figure 2. Couverture de code

L'exemple effectue un parcours en profondeur (*backtracking* ou *deep first search*) sur l'arbre d'exécution. Ce n'est pas le seul parcours pouvant être adopté. Par ailleurs, le choix d'un parcours pour une couverture optimale du code est encore un problème ouvert (« problème de la sélection de chemin » [15]). *SAGE* propose le *Generational Search* [14] afin de maximiser le nombre d'entrées générées pour chaque trace exécutée symboliquement.

2.4 Exécution symbolique statique (SSE)

Comme dit précédemment, la SSE est une technique de vérification. Elle représente le programme tout entier par une formule logique. Les vulnérabilités potentielles quant à elles sont représentées par des assertions logiques. Si une vulnérabilité est découverte, celle-ci violera une assertion rendant fausse la formule du programme.

L'avantage principal du SSE est de travailler directement sur tout le programme (l'ensemble des chemins possibles). De ce fait, les formules produites par du SSE sont plus générales que celles du DSE. De plus, les algorithmes modernes de SSE et certaines techniques permettent de générer des formules compactes en taille (quadratique par rapport au nombre d'instruction) pour des programmes exempt de boucle [1]. Cette approche soulève néanmoins certaines questions comme :

- Combien de fois doit-on dérouler une boucle ?
- Les formules peuvent-elle être résolues par les SMT solver si celles-ci prennent en compte les boucles et la récursivité ?
- Comment modéliser correctement un appel système ?

À cause notamment de ces problèmes, le SSE ne s'adapte pas bien au programme très large. Néanmoins, des solutions telles que *Veristesting* [1] tentent de combiner l'utilisation du SSE et du DSE. Dans *Veristesting*, le SSE est utilisé au sein d'une exécution dynamique, sur des portions de programme acyclique, afin d'obtenir une plus large couverture de code en une seule passe.

2.5 Utilisation de l'instrumentation binaire pour la récupération des instructions exécutées

L'instrumentation binaire aussi appelée DBI (*Dynamic Binary Instrumentation*) est maintenant une approche fréquemment utilisée pour analyser le comportement d'un binaire. Elle consiste en l'ajout d'instructions en cours d'exécution sans toucher au code d'origine. Ainsi, il est possible d'ajouter des fonctions d'analyses (*callbacks*) avant et/ou après chaque instruction exécutée. Cela permet d'avoir une vision de ce qui est exécuté, à quelle adresse, avec quel type d'opérande, leurs contenus, les zones mémoires qui sont lues ou écrites, etc...

Ce contexte concret est utile lors d'une exécution symbolique dynamique afin de créer la trace sémantique. Il nous permet de savoir quelles

sont les instructions exécutées et de les traduire en leur expression sémantique avec leurs effets de bord comme les EFLAGS. Cette partie sera décrite plus en détail dans les sections qui concernent le fonctionnement interne de Triton.

2.6 Utilisation de Pin

Il existe plusieurs outils d'instrumentation mais notre choix s'est tourné sur l'utilisation de Pin [9] pour sa simplicité d'utilisation. Le projet aurait pu être implémenté avec ou au-dessus de n'importe quel autre outil de DBI car la méthode employée n'est pas dépendante du moteur d'instrumentation.

Pin est un framework développé par Intel pour le jeu d'instructions x86 et x86-64 qui permet la création d'outils d'analyse dynamique. Les outils développés avec Pin s'appellent des Pintools et sont Intel Parallel Inspector [7], Intel Parallel Advisor [6] et Intel VTune Amplifier [8].

Le fait d'instrumenter un binaire a un impact évident sur son temps d'exécution. Ce coût est variable en fonction de l'outil de DBI utilisé ainsi que de la modélisation des analyses. La Figure 3 illustre cette surcharge sur trois cas : une exécution sans instrumentation, une exécution avec une *callback* sur chaque instruction sans l'envoi du contexte [10] et enfin, une exécution avec une *callback* sur chaque instruction avec l'envoi du contexte.

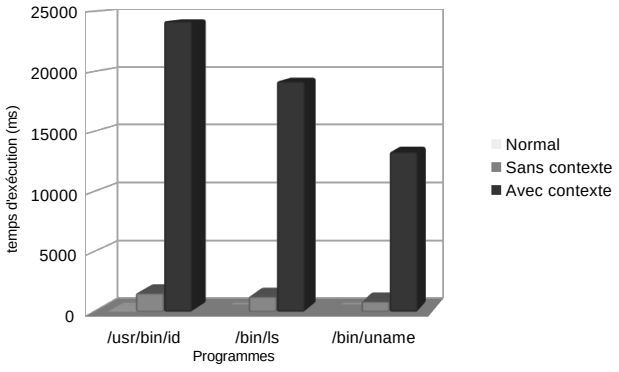


Figure 3. Overhead avec Pin

Bien qu'il y ait un surcoût en temps d'exécution assez important, Pin propose tout de même une couche d'abstraction pour manipuler l'instrumentation et permet d'arriver rapidement à des Pintool intéressants en très peu de lignes de code.

3 Triton

Nous commencons par exposer le fonctionnement global de Triton avant d'expliquer le traitement spécifique effectué par chaque « moteur ». Triton se compose de 3 composants principaux interagissant entre eux :

- L'instrumentation dynamique (DBI) à travers Pin. Son rôle est de récupérer le contexte concret associé à l'instruction courante puis de le transmettre au cœur.
- Le cœur réalise les analyses (*taint analysis* et exécution symbolique), récupère les contraintes, génère les prochaines entrées... Elle contient tous les « moteurs » ainsi que les analyses axées sur la recherche de vulnérabilités.
- Le SMT solver Z3 traite les formules envoyées par le cœur. Son rôle est de résoudre dans la théorie Bit Vector ces formules. Si elles sont satisfaisable celui renvoie au cœur un résultat (un élément du modèle).

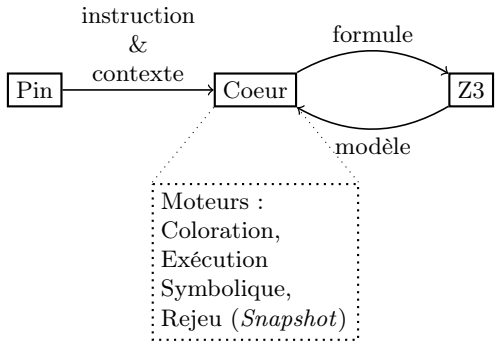


Figure 4. Schéma global des composants de Triton.

3.1 Moteur de teinte

Le rôle du moteur de teinte est de mémoriser les registres et les adresses mémoires *teintés* tout au long d'une trace.

Cette analyse présente trois intérêts au sein de Triton :

Son premier rôle est d'aider l'exécution symbolique à décider si la valeur d'une adresse mémoire doit être traitée symboliquement ou concrètement. Typiquement, si l'opérande source d'une instruction n'est pas *teintée*, la valeur concrète est suffisante. Dans le cas contraire, la valeur actuelle dépend des entrées spécifiées par l'utilisateur. Triton utilise donc la valeur symbolique de l'opérande source.

Le second rôle est un cas particulier. La *taint analysis* et l'exécution symbolique travaillent toutes les deux sur les mêmes entrées. Pour la première, elles représentent les sources d'où se propage la *teinte*. Pour la seconde, ces cases mémoires *teintées* sont les variables symboliques sur lesquelles l'exécution va porter. L'exécution symbolique va se baser sur la *teinte* de la mémoire afin de construire les variables symboliques. Une case mémoire *teintée* mais inconnue du moteur symbolique est le signe d'une nouvelle entrée, et donc une variable symbolique la représentant doit être créée.

Enfin, la *teinte* peut aussi servir de filtre pour l'exécution symbolique. Au cours d'une trace, un grand nombre d'instructions n'agit pas sur les entrées spécifiées par l'utilisateur (aucun de leurs opérandes ne sont *teintés*). Ces instructions n'apparaîtront donc jamais dans les contraintes du *Path Condition*. Il n'est pas utile alors de les exécuter symboliquement. Cette optimisation de l'exécution symbolique grâce au *taint analysis* est utilisée au sein de *MergePoint* [1] et de *FuzzWin* [12].

3.2 Moteur de rejeu (*Snapshot*)

Pin est capable d'instrumenter une trace mais également de modifier son contexte ou son flux d'exécution en *runtime* [13]. Le moteur de rejeu utilise cette capacité de Pin afin de remonter la trace. Son rôle est de prendre, puis de stocker un instantané du contexte d'exécution (état des registres et de la mémoire) pour chaque branchement trouvé.

Afin de restaurer le contexte d'exécution (concret et symbolique) correctement, il est nécessaire de sauvegarde à chaque saut conditionnel l'état des registres concrets et symboliques. Par ailleurs, pour réellement remonter la trace d'exécution, il faut aussi restaurer l'état de la mémoire concrète et symbolique au moment du saut.

Pour cela, Triton maintient une liste des modifications de la mémoire. Chaque modification est sauvegardée au sein d'un triplet : $\langle$adresse, valeur concrète, valeur symbolique$\rangle$. A chaque écriture dans la mémoire, nous rajoutons à cette liste un nouveau triplet permettant de sauvegarder les anciennes valeurs.

Lors d'un saut conditionnel, il suffit d'associer la taille actuelle de cette liste au saut en question. Il sera utilisé en tant qu'indice lors de la restauration.

Triton réalise cette restauration en deux étapes. Tout d'abord, il remplace l'état de ses registres symboliques par la sauvegarde correspondant au saut où il souhaite revenir. Il demande à Pin de faire de même. Ainsi les registres sont restaurés. Puis il s'occupe de la mémoire. Pour cela, il parcoure la liste de modifications en sens inverse jusqu'à l'indice associé au saut, tout en restaurant les valeurs sauvegardées (concrètes et symboliques encore une fois) aux adresses correspondantes.

Le binaire analysé peut agir sur l'environnement externe, comme par exemple créer ou écrire dans un fichier. Ces modifications là ne sont pas restaurées par Triton. En effet, l'ensemble des actions sur l'environnement externe ne sont connues du binaire qu'à travers leurs résultats (valeur de retour des appels systèmes). Ces informations sont véhiculées au sein du programme grâce aux registres et à la mémoire que Triton peut modifier. À la différence d'une machine virtuelle qui restaure réellement tout un système de fichiers, Triton ne modifie que le contexte d'exécution en cours.

3.3 Moteur symbolique

Le rôle du moteur symbolique est de mémoriser les valeurs symboliques associées aux registres et à la mémoire lors de l'exécution d'une trace. Concrètement, pour les registres, on utilise tableau où chaque case représente la référence symbolique d'un registre (Tableau 2). En ce qui concerne le registre particulier des EFLAGS, chaque flag (AF, CF, OF, PF, SF, ZF) possède une case qui lui est propre dans ce tableau. Dans un environnement multi-threadé, il y aura un tableau par thread. Pour les références sur la mémoire, cela est géré par une map $\langle \Delta : R \rangle$, où Δ est l'adresse mémoire de la référence et R un nombre entier qui représente son ID unique.

Registre	Référence ID
RAX	#47
RBX	#31
RCX	UNSET
RDX	#33
RDI	#13
RSI	UNSET
...	...
AF	#48
PF	#49

Table 2. Table de références registre/symbole ID

3.4 Sémantique et représentation intermédiaire

Triton transforme chaque instruction assembleur en une liste d'une ou plusieurs formules SMT2-LIB. Ce langage et son intérêt sont présentés dans le paragraphe suivant 3.5, mais sa connaissance n'est pas nécessaire à la compréhension de ce qui suit. Une formule SMT2-LIB est une *S-expression*, sa syntaxe ressemble fortement au langage Lisp ou Scheme. Toutes les opérations sont préfixées et une expression bien-formée est comprise dans des parenthèses. Chaque formule dispose d'un identifiant unique (ID) la représentant. Si une formule A dépend de la valeur d'une formule B, elle utilisera l'ID de B en lieu et place de l'autre formule. Ce principe de l'identifiant unique provient du « *Single Static Assignment form* » ou SSA. Dans la suite de ce papier, nous appellerons ces formules ainsi que cette représentation intermédiaire, « SSA-expression ». Sa grammaire est donnée dans la Table 3.

Par exemple, le Listing 2 présente les formules correspondant à l'addition de deux registres. Notez que celui-ci est incomplet pour des raisons de lisibilité. En effet, l'ensemble des formules représentant la valeur symbolique des EFLAGS a été omis.

```
mov eax, 0x15   #1  = (_ bv21 32)
mov ebx, 0x32   #2  = (_ bv50 32)
add eax, ebx    #3  = (bvadd #1 #2)
```

Listing 2. Formule issue de l'addition de deux registres

Triton stocke les SSA-expressions sous la forme de chaînes de caractères. Les avantages de cette représentation sont les suivants :

Stat	::=	*ID* "=" *Expr*
Expr	::=	"(" *BinOp Expr Expr* ")" |
		"(" *UnOp Expr* ")" |
		ID
BinOp	::=	"extract" | "bvadd" | "bvsub" | "bvand" | ...
UnOp	::=	"(_" *Ext* ")"
Ext	::=	"sign-extend" | "zero-extend"
ID	::=	"#"*Entier*

Table 3. Grammaire d'une SSA-expression.

– construction facilitée de la formule : Il suffit de remplacer récursivement toutes les références vers une sous-formule par leur valeur ;

– génèration directement d'un code SMT2-LIB valide pouvant être passé au SMT solver.

L'inconvénient de cette représentation est sa rigidité. En effet, la représentation sous forme de chaîne de caractères complique les simplifications pouvant être effectuées au sein d'une formule. C'est par exemple le cas pour : remplacer le résultat issue de l'addition de deux valeurs exactes (*constant folding*[5]) ou supprimer une contrainte redondante (situation courante dans le cas d'une boucle). Actuellement, Triton n'effectue donc pas de simplification sur les contraintes envoyées au SMT solver. Néanmoins, comme le montrent les résultats sur *SAGE* et *MergePoint* [14,1], ces simplifications sont nécessaires afin que des programmes de taille conséquente puissent être gérés.

3.5 Langage SMT2-LIB

Le langage SMT-LIB [16] version 2 (que nous appelons SMT2-LIB) est la poursuite d'une initiative internationale visant à faciliter la recherche et le développement sur le problème de *Satisfaisabilité Modulo Théorie (SMT)*.

Cette nouvelle version est supportée par un bon nombre SMT solver. De ce fait, traduire nos contraintes dans ce langage plutôt que d'utiliser l'interface `C++` de Z3, permet de découpler notre *Cœur* du *SMT solver*.

3.6 Résolution des contraintes avec Z3

Triton embarque une classe *SolverEngine* qui fait office de lien entre Triton et Z3. Cela permet de résoudre certaines contraintes comme, par

exemple, la résolution d'une formule permettant de prendre la branche *Vrai* (V) ou *Faux* (F) lors d'une instruction de branchement.

Le Listing 3 montre un extrait d'une trace d'exécution au moment d'une instruction de branchement.

```
...
movsx eax, al       #49 = ((_ sign_extend 24) #48)
cmp ecx, eax        #50 = (bvsub #38 #49)
                    ... ; Tous les autres flags
                    #56 = (assert (= #50 (_ bv0 32))) ; ZF
jz 0x4005b9         b(#56)
```

Listing 3. Extrait d'une trace d'exécution

Au moment du branchement, la référence du drapeau `ZF` est 56. Cet ID fait référence à l'expression `(assert (#50 (_ bv0 32)))`. Cette formule peut être envoyée au moteur de résolution de contraintes (*SolverEngine* (SE)), pour pouvoir générer un modèle valide qui nous permettra de prendre soit la branche V ou F.

Le SE s'occupera d'effectuer du « *backward reconstruction* » pour reconstruire une formule qui sera dite « valide » pour un *theorem prover*. Comme on peut le voir sur l'expression précédente du ZF, elle contient la référence #50. Cette référence sera remplacée par son expression et ainsi de suite récursivement jusqu'à ce qu'il n'y ait plus de référence. Le Listing 4 illustre ce procédé.

```
Before Backward Reconstruction
------------------------------
Formula = "(assert (= #50 (_ bv0 32)))"

After Backward Reconstruction
-----------------------------
Formula = "(assert (= (bvsub (bvxor (bvsub ((_ sign_extend 24) ((_
    extract 7 0)((_ zero_extend 24) SymVar_0)))(_ bv1 32)) (_ bv85
    32)) ((_ sign_extend 24) ((_ extract 7 0)((_ zero_extend 24) (_
    bv49 8))))) (_ bv0 32)))"
```

Listing 4. Formule reconstruite

La formule reconstruite est ensuite donnée à *Z3* qui renvoit un modèle valide. Il est également possible d'effectuer du *fuzzing* sur un ou des chemins précis en demandant tous les modèles qui satisfont la les conditions de branchement.

Imaginons que nous voulions *fuzzer* le paramètre x de la fonction f dans le Listing 5. Au lieu de tester les 2^{32} possibilités que peut contenir la

variable x, il est possible d'effectuer une première passe symbolique qui permettra de créer l'expression symbolique de la condition de branchement `x > 1000` $\wedge$ `x` $\leq$ `1050`. Lors de la deuxième itération, la formule symbolique est envoyée au SE qui retourne un modèle valide permettant de passer dans le branchement ligne 4. Le procédé a été expliqué précédemment avec la Figure 2.

Le fait de demander au SE tous les modèles possibles que peut prendre cette branche, permet d'effectuer du *fuzzing* symbolique et de ne pas perdre de temps à affecter à x des valeurs aléatoires différentes mais qui ne permettent pas toujours d'aller dans cette branche. Cela réduit ainsi le champ des valeurs possibles de 2^{32} à 50 qui est le cardinal du modèle.

```
1. uint32_t f(uint32_t x)
2. {
3.     if (x > 1000 && x <= 1050){
4.         ...
5.     }
6.     return -1;
7. }
```

Listing 5. Fuzzing et exécution symbolique

3.7 Multi-threads

Pin a une très bonne gestion des threads ce qui a permis d'implémenter la gestion du multi-threads sur le moteur symbolique et celui de teinte très facilement. Lors de l'instrumentation, Pin a la possibilité de fournir pour chaque *callback* le numéro du thread (*THREADID*) dans lequel est exécutée l'instruction. Pour la classe symbolique, il existe une table de référence pour chaque thread en cours et lors de l'instrumentation, le THREADID est utilisé comme index pour accéder à la bonne table. La Figure 5 illustre ce procédé.

Sur la Figure 5, la classe du moteur symbolique (1) maintient un unique ensemble d'expressions symboliques tous threads confondus (2) [2]. Cela permet au solver de contraintes de pouvoir accéder à l'ensemble des expressions facilement et rapidement. Ensuite, chaque thread a sa propre table de références (4) qui fait le lien $registre \longleftrightarrow ID$ (Tableau 2) de son thread.

La gestion de la teinte est similaire, il y a un tableau par thread accessible par le THREADID comme index.

2. Pour rappel, ces expressions sont illustrées par le Listing 3.

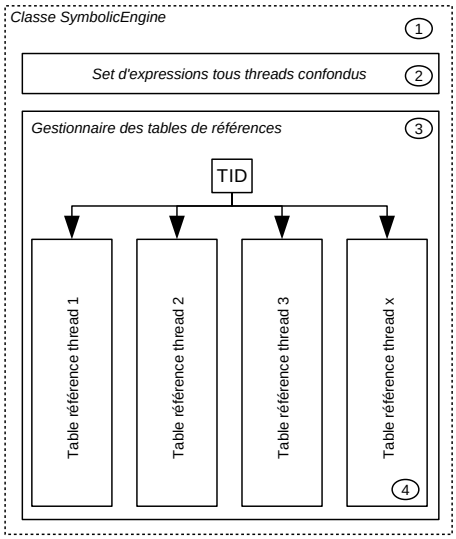


Figure 5. Moteur symbolique et multi-thread

3.8 Bindings Python

Le défi pour un framework d'exécution concolique est d'être le plus *user-friendly* possible. C'est pourquoi nous avons exporté l'API de Triton depuis des bindings Python, ce qui évite à l'utilisateur de développer directement ses analyses en `C++` dans le code de Triton.

L'idée est de développer ses analyses en Python dans un fichier externe et d'envoyer ce fichier en argument au Pintool Triton comme illustré dans le Listing 6

```
pin -t ./triton.so -script your_tool.py -- your_target_binary
```

Listing 6. Utilisation de Triton

Depuis les bindings Python il est possible de récupérer des informations à chaque point du programme comme la *teinte* d'une adresse mémoire ou d'un registre, son état symbolique, les expressions symboliques...

```
from triton import *


def my_callback(instruction):

    print '%#x: %s' %(instruction.address, instruction.assembly)

    for se in instruction.symbolicElements:
```

```
        print '\t -> ', se.expression


if __name__ == '__main__':

    startAnalysisFromSymbol('check')

    taintRegFromAddr(0x40058e, [IDREF.REG.RAX, IDREF.REG.RBX])

    addCallback(my_callback, CB_AFTER)

    runProgram()
```

Listing 7. Exemple d'outil Triton

L'outil Triton illustré par Listing 7, permet de lancer une exécution symbolique sur la fonction *check*, de teinter le registre `RAX` et `RBX` à l'instruction `0x40058e` et d'afficher les expressions symboliques à chaque instruction. La sortie de cet outil est illustré par le Listing 8.

```
[...]
0x400591: sub eax, 0x1
          ->  #24 = (bvsub #23 (_ bv1 32))
          ->  #25 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor
              #24 (bvxor #23 (_ bv1 32))))))
          ->  #26 = (assert (bvult #24 #23))
          ->  #27 = (assert (= ((_ extract 31 31) (bvand (bvxor #23 (
              bvnot (_ bv1 32))) (bvxor #23 #24))) (_ bv1 1)))
          ->  #28 = (assert (= (parity_flag ((_ extract 7 0) #24)) (_
               bv0 1)))
          ->  #29 = (assert (= ((_ extract 31 31) #24) (_ bv1 1)))
          ->  #30 = (assert (= #24 (_ bv0 32)))
0x400594: xor eax, 0x55
          ->  #31 = (bvxor #24 (_ bv85 32))
          ->  #32 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor
              #31 (bvxor #24 (_ bv85 32))))))
          ->  #33 = (assert (bvult #31 #24))
          ->  #34 = (assert (= ((_ extract 31 31) (bvand (bvxor #24 (
              bvnot (_ bv85 32))) (bvxor #24 #31))) (_ bv1 1)))
          ->  #35 = (assert (= (parity_flag ((_ extract 7 0) #31)) (_
               bv0 1)))
          ->  #36 = (assert (= ((_ extract 31 31) #31) (_ bv1 1)))
          ->  #37 = (assert (= #31 (_ bv0 32)))
0x400597: mov ecx, eax
          ->  #38 = ((_ extract 31 0) #31)
[...]
```

Listing 8. Exemple de sortie d'outil Triton

4 Les analyses en *runtime*

Lors de la recherche de vulnérabilités, la couverture de code ne garantit pas de trouver tous les bugs présents dans le programme. Dans beaucoup

de cas, les bugs ne causent pas de crash sur le moment. Les crashs peuvent se produire bien plus tard dans la trace d'exécution.

L'idée des analyses dans Triton est de pouvoir détecter en *runtime* la présence de « comportements suspects » pouvant amener un bug potentiel sans se baser sur les crashs. Le principe est d'utiliser l'exécution symbolique pour couvrir un maximum de chemins, puis d'y appliquer les analyses en *runtime*. Si un crash se produit ou que les analyses détectent un bug potentiel, le moteur symbolique nous permet de générer et de rejouer les bonnes entrées pour arriver sur la partie de code « suspecte ».

Il est difficile de mettre en place une méthode générique permettant de trouver tous type de bugs. C'est pourquoi il est nécessaire d'appliquer, pour chaque catégorie de bugs, une analyse qui lui est propre. Ces analyses seront détaillées dans les paragraphes suivants par catégorie de bugs.

4.1 Analyse pour détecter les *formats strings*

Pour la détection de potentielles vulnérabilités de type *FormatString*, nous utilisons le moteur de teinte pour connaître à chaque point du programme quels sont les registres contrôlés par l'utilisateur.

Une vulnérabilité de type *FormatString* est présente lors d'un appel à *printf* (ou autre fonction dérivée) avec comme premier argument une chaîne contrôlable par l'utilisateur. Généralement, le premier argument se doit d'être une constante contenant une chaîne de format (`%s, %x, %02x`, ...).

Nous savons également que sur une architecture Intel x86-64, la *calling convention* principalement utilisée est `__fastcall`. Triton va donc vérifier pour chaque appel à *printf* (ou ses dérivées) si le registre `RDI` au moment de l'appel est marqué comme contrôlable ou partiellement contrôlable par le moteur de teinte. Si c'est le cas, cela peut potentiellement être une vulnérabilité de type *FormatString*. Pour s'assurer de la présence d'une telle vulnérabilité, il est possible de demander au moteur symbolique d'extraire la formule symbolique permettant d'accéder à ce chemin afin d'effectuer une vérification manuelle.

4.2 Analyse pour détecter les *use-after-free*

Les *use-after-free* sont des vulnérabilités de plus en plus courantes. Elles se concrétisent par l'utilisation (STORE/LOAD) d'un pointeur pointant sur une zone mémoire libérée (voir Figure 6).

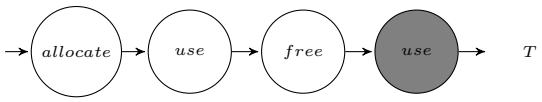


Figure 6. use-after-free

Pour détecter ce type de comportement, Triton surveille les appels à *malloc* et à *free* puis maintient une table des zones mémoires allouées (TA) ainsi qu'une table des zones libérées (TF). Ces tables sont composées de couples : $\langle \Delta, S \rangle$ où Δ est la base et S la taille. Chaque couple correspond à une zone mémoire allouée ou libérée selon si elle se trouve dans TA ou TF.

Dans TA, Δ est obtenu par le retour de *malloc* (`RAX`) et S est obtenu par le premier argument de *malloc* (`RDI`). Lors d'un appel à *malloc*, l'élément $\langle \Delta, S \rangle$ est créé, puis placé dans TA. Lors d'un appel à la fonction *free*, Δ est obtenu par le premier argument de la fonction (le registre `RDI`), puis le Δ correspondant dans la table TA est déplacé dans TF.

Lors d'un appel à *malloc* avec un Δ correspondant à une entrée dans TF, si $S_{old} = S_{new}$ alors l'élément TF est déplacé dans TA, dans le cas contraire l'élément dans TF est divisé en deux nouvelles zones.

Triton surveille également toutes les instructions effectuant des STORE et LOAD puis vérifie si l'adresse mémoire correspond bien à une adresse située dans TA ou TF sur trois critères.

- si $\Delta \in TA \rightarrow$ Accès mémoire valide ;
- si $\Delta \notin TA \wedge \Delta \notin TF \rightarrow$ Accès mémoire invalide ;
- si $\Delta \notin TA \wedge \Delta \in TF \rightarrow$ Use-after-free.

4.3 Analyse pour détecter les débordements de tampons dans le tas

Le fonctionnement interne de cette analyse est très similaire à celle utilisée pour les *uses-after-free*. Une vulnérabilité de type débordement de tampon sur le tas a généralement lieu lorsqu'une boucle effectue un STORE linéaire (Figure 7) dans un *buffer* ayant été alloué dans le tas et que l'écriture dépasse de la zone mémoire allouée.

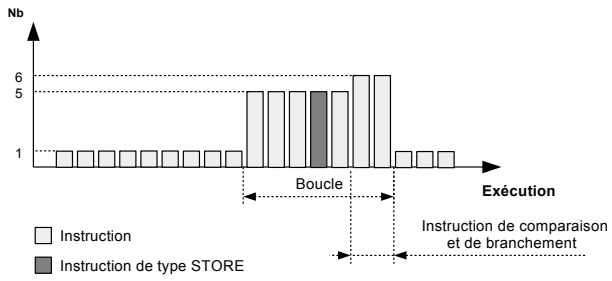


Figure 7. Schéma d'une boucle qui effectue un STORE

Plusieurs types d'instructions peuvent être effectuées sur une zone mémoire (STORE/LOAD - Figure 8). Dans cette analyse, Triton traite les STORE et LOAD identiquement et comme pour les *uses-after-free* il maintient une table d'allocation mémoire (TA). Chaque élément dans TA est un couple $\langle \Delta, S \rangle$ où Δ est l'adresse de base de l'allocation et S la taille de cette zone.

Lors d'une boucle, s'il y a une instruction de type LOAD ou STORE et que la lecture ou l'écriture commence dans la zone allouée et déborde de cette zone, Triton reconnaîtra cela comme une vulnérabilité de type *débordement de tampon dans le tas* (même si la boucle n'effectue qu'un LOAD). Contrairement à l'analyse pour les *uses-after-free* où la première lecture ou écriture a lieu sur une zone n'étant plus allouée.

On note $\beta \in \mathbb{N}_{\geq 0}$ le nombre d'itérations lors d'une boucle et $\langle \Delta_\beta, S_\beta \rangle$ le tuple correspondant à la zone mémoire pour chaque tour de boucle. Lors d'un STORE/LOAD nous appliquons les 3 règles suivantes :

- si $\Delta \notin TA \rightarrow$ Accès invalide ;
- si $\Delta \in TA \wedge \beta \in \mathbb{N}_{>0} \wedge \Delta_\beta = \Delta_{\beta-1} \wedge \beta < S \rightarrow$ Accès valide ;
- si $\Delta \in TA \wedge \beta \in \mathbb{N}_{>0} \wedge (\Delta_\beta \neq \Delta_{\beta-1} \vee \beta \geq S) \rightarrow$ Heap overflow.

4.4 Analyse pour détecter les débordements de tampons sur la pile

Bien que les débordements de tampons sur la pile et dans le tas soient très similaires d'un point de vue théorique (débordement d'un espace alloué), l'approche pour détecter les débordements sur la pile est bien plus complexe que celle pour détecter les débordements dans le tas.

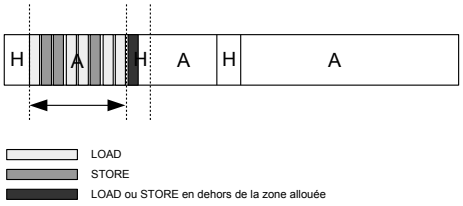


Figure 8. Tampon dans le tas et accès mémoire

La grande difficulté n'est pas de vérifier si une écriture a lieu en dehors d'une *stack frame* mais de vérifier la présence d'un débordement entre variables d'une même *stack frame*. Lors d'une allocation mémoire sur le tas, nous savons qu'un buffer est généralement utilisé pour une variable (sauf cas exceptionnel), alors que l'allocation faite par le prologue d'une fonction alloue un espace mémoire pour toutes les variables locales. La difficulté est donc de savoir combien de variables sont présentes sur une *stack frame*.

Pour illustrer ce problème, prenons comme exemple le code du Listing 9. Dans cet exemple, nous avons trois variables locales puis une boucle vulnérable à un *off-by-one* qui effectue un STORE sur la variable a. Ce type de bug ne provoque pas le crash du programme mais est bel et bien présent. L'objectif est de faire en sorte que Triton puisse détecter en *runtime* ce type de bug, dès lors qu'il y a un débordement entre variables.

```
uint32_t a, b, c;

a = 0x11111111;
b = 0x22222222;
c = 0x33333333;

for (uint32_t i = 0; i <= sizeof(a); i++) /* off-by-one */
    *(((unsigned char *)(&a))+i) = 0xff;
```

Listing 9. Exemple d'un off-by-one

La première étape est d'isoler les fonctions et d'assigner un ID unique pour chaque nouvelle *stack frame*. Nous devons ensuite détecter le nombre de variables présentes sur la *stack frame* ainsi que leurs tailles. Chaque variable est modélisée par un triplet $\langle I, \Delta, S \rangle$ où $I \in \mathbb{N}_{\geq 0}$ représente l'ID unique de la *stack frame*, Δ, l'adresse de la variable dans cette *stack frame*, et $S \in \mathbb{N}_{>0}$ la taille de la variable.

Pour l'isolation des fonctions, Triton se base sur les routines fournies par Pin. Pour le nombre de variables dans une *stack frame*, on reprend

le même principe que les *A-Locs* présentées dans l'article [2] qui parle du VSA (*Value-Set Analysis*).

Le Listing 10 illustre un court *dump* de plusieurs *stack frame ID* avec leur nombre de variables et leurs adresses. Si deux assignations sont effectuées sur un même $\langle I, \Delta, S \rangle$ (comme à la ligne 7 et 9), Triton ne prend pas en compte cela comme deux variables mais comme une variable ayant reçu deux assignations. Cependant, si la deuxième assignation a lieu sur $\langle I, \Delta + n, S \rangle$, Triton raffine le triplet en le divisant en deux variables distinctes.

```
01. 4006d4: mov qword ptr [rbp-0x10], 0x0
02.     -> (dest: 7fffdae70d10) (stack frame ID: 3)
03. 4006dc: mov qword ptr [rbp-0x18], 0x0
04.     -> (dest: 7fffdae70d08) (stack frame ID: 3)
05. 4006e4: mov dword ptr [rbp-0x4], 0x0
06.     -> (dest: 7fffdae70d1c) (stack frame ID: 3)
07. 40071f: mov dword ptr [rbp-0x4], 0x0
08.     -> (dest: 7fffdae70d1c) (stack frame ID: 5)
09. 400742: mov dword ptr [rbp-0x4], 0x0
10.     -> (dest: 7fffdae70d1c) (stack frame ID: 5)
11. 400640: mov dword ptr [rbp-0x4], 0x0
12.     -> (dest: 7fffdae70cdc) (stack frame ID: 9)
13. 400669: mov dword ptr [rbp-0x8], 0x90909090
14.     -> (dest: 7fffdae70cd8) (stack frame ID: 10)
15. 400670: mov dword ptr [rbp-0xc], 0x91919191
16.     -> (dest: 7fffdae70cd4) (stack frame ID: 10)
17. 400677: mov dword ptr [rbp-0x4], 0x0
18.     -> (dest: 7fffdae70cdc) (stack frame ID: 10)
20.
21. Stack frame ID: 3    Num var: 3
22. Stack frame ID: 5    Num var: 1
23. Stack frame ID: 9    Num var: 1
24. Stack frame ID: 10   Num var: 3
```

Listing 10. Dump du nombre de variables par stack frame

Lors d'une boucle effectuant un STORE linéairement [3] (Figure 7), si Triton constate un changement de zone basé sur sa liste de $\langle I, \Delta, S \rangle$ durant l'itération, il reconnaît cela comme étant un débordement de tampon sur la pile.

On note $\beta \in \mathbb{N}_{\geq 0}$ le nombre d'itération lors d'une boucle et $\langle I_\beta, \Delta_\beta, S_\beta \rangle$ le triplet correspondant à la zone mémoire pour chaque tour de boucle. Lors d'un STORE/LOAD nous appliquons les 3 règles suivantes :

3. Boucle qui effectue un STORE sur $\Delta + \beta$ où Δ est la base constante et β le nombre de tours de boucles effectués à chaque itération.

– si $\beta \in \mathbb{N}_{>0} \wedge I_\beta = I_{\beta-1} \wedge \Delta_\beta = \Delta_{\beta-1} \rightarrow$ Accès valide ;
– si $\beta \in \mathbb{N}_{>0} \wedge I_\beta \neq I_{\beta-1} \rightarrow$ *Overflow* en dehors de la stack frame ;
– si $\beta \in \mathbb{N}_{>0} \wedge \Delta_\beta \neq \Delta_{\beta-1} \rightarrow$ *Overflow* entre deux variables.

Dans le cas du Listing 9, le schéma d'écriture est illustré par la Figure 9 et nous constatons un changement de zone sur la dernière itération (I_1 : `[3,0x7fffffffdc10,4]` $\rightarrow$ I_2 : `[3,0x7fffffffdc10,4]` $\rightarrow$ I_3 : `[3,0x7fffffffdc10,4]` $\rightarrow$ I_4 : `[3,0x7fffffffdc10,4]` $\rightarrow$ I_5 : $[3, \mathbf{0x7fffffffdc14}, 4]$).

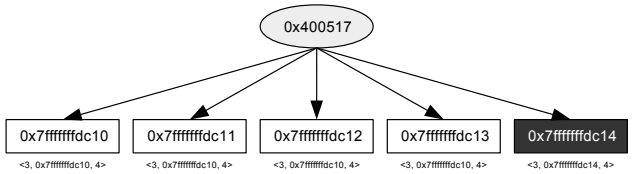


Figure 9. Off-by-one - Changement de zone $\langle I, \Delta, S \rangle$ sur la dernière itération

Le point négatif de cette implémentation est la génération de « faux positifs » par le type de code illustrés dans le Listing 11 qui sont souvent utilisés pour initialiser tous les attributs d'une structure à une valeur donnée. Imaginons $A, B \in S$ tel que $A \rightarrow \langle 1, \Delta, 4 \rangle$, $B \rightarrow \langle 1, \Delta + 4, 4 \rangle$ et $S \rightarrow \langle 1, \Delta, 8 \rangle$. Dans ce cas, Triton détectera des débordements entre variables d'une même *stack frame*, ce qui est vrai mais voulu par le programmeur.

```
struct s_foo S;

memset(&S, 0, sizeof(struct s_foo));
...
```

Listing 11. Exemple de faux positif sur les changements de zones $\langle I, \Delta, S \rangle$

4.5 Analyse pour détecter les *{write, read}-what-where*

Cette analyse se base principalement sur le moteur de teinte pour savoir si à un instant T, les registres de l'instruction courante sont contrôlables par l'utilisateur. L'analyse surveille seulement les instructions de type LOAD et STORE.

– Cas 1 : `mov [x], y`
– Cas 2 : `mov x, [y]`

Dans le cas 1, si à n'importe quel moment de l'exécution, le registre x est marqué comme étant teinté, cela signifie que nous contrôlons l'adresse de destination (*write-where bug*). Toujours dans le même cas, si x et y sont marqués comme teinté, nous avons une vulnérabilité de type *write-what-where*. Sur le même principe, dans le cas 2, si nous contrôlons y, nous avons une lecture arbitraire de la mémoire. La Figure 10 illustre cette analyse.

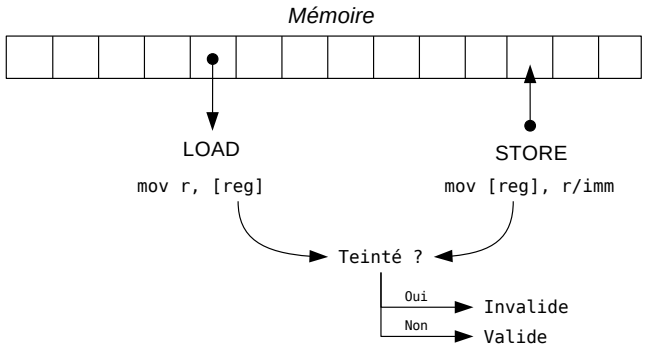


Figure 10. Analyse des vulnérabilités de type {write, read}-what-where

Ensuite, il est possible d'utiliser le moteur symbolique pour connaître quel est le *Path Condition* pour arriver sur cette instruction et pour connaître le *set* de valeurs possibles que peut contenir x ou y au moment de la vulnérabilité.

5 Conclusion

Triton est encore un jeune projet proposant des composants supplémentaires au framework Pin. Ces composants permettent d'effectuer de l'analyse par teinte, de générer des traces symboliques, de représenter les sémantiques en SMT2-LIB, de rejouer des traces à l'aide d'un moteur de snapshot et de stocker ces informations dans une base de données externe.

L'API de Triton peut être utilisée pour de la recherche de vulnérabilités, de l'analyse de traces, du test logiciel (*fuzzing*) ou encore d'aider un *reverse engineer* pendant ses passes d'analyses statiques.

Références

1. 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.
2. Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.
3. Gabriel Campana. Fuzzgrind : un outil de fuzzing automatique. SSTIC, 2009.
4. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
5. Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
6. Intel. Intel parallel advisor. http ://en.wikipedia.org/wiki/Intel_Parallel_Advisor.
7. Intel. Intel parallel inspector. http ://en.wikipedia.org/wiki/Intel_Parallel_Inspector.
8. Intel. Intel vtune amplifier. http ://en.wikipedia.org/wiki/VTune.
9. Intel. Pin - a dynamic binary instrumentation tool. https ://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
10. Intel. Pin - reference : Context. http ://goo.gl/xpE2TK.
11. James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.
12. Sébastien Lecomte. Élaboration d'une représentation intermédiaire pour l'exécution concolique et le marquage de données sous windows. SSTIC, 2014.
13. Heidi Pan, Robert Cohn Krste Asanovic, and Chi-Keung Luk. Controlling program execution through binary instrumentation. http ://www.cs.berkeley.edu/ krste/papers/pin-wbia.pdf.
14. Michael Y. Levin Patrice Godefroid and David Molnar. Sage : Whitebox fuzzing for security testing. 2012.
15. 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.
16. SMT-LIB. Site web de la smt-lib. http ://smt-lib.org/.