# Élaboration d'une représentation intermédiaire pour l'exécution concolique et le marquage de données sous Windows

Sébastien Lecomte
cyberaware@laposte.net

**Résumé** Alors que sous GNU/Linux, Valgrind offre au programmeur une représentation intermédiaire pour instrumenter dynamiquement un binaire, il n'existe pas d'outil aussi complet sur les OS Microsoft.

Cette problématique a été rencontrée lors de la tentative de portage sous Windows du fuzzer *Fuzzgrind*, développé par Gabriel Campana sur plateforme GNU/Linux et présenté lors de l'édition 2009 du SSTIC [3]. *Fuzzgrind* est basé sur une exécution concolique du code binaire, laquelle nécessite l'emploi impératif d'un langage intermédiaire.

Le projet de portage de *Fuzzgrind*, baptisé *FuzzWin*, a donc nécessité l'écriture d'un langage intermédiaire et un ensemble de fonctions spécifiquement développées par l'auteur pour le marquage de données. Après avoir rapidement rappelé le fonctionnement et l'algorithme du fuzzer, cet article se propose de détailler les grandes caractéristiques de ce langage ainsi que son utilisation dans le projet *FuzzWin*.

**Mots-clés:** instrumentation de binaire, représentation intermédiaire, marquage de données, exécution concolique.

## 1 Fonctionnement général de FuzzWin

### 1.1 L'algorithme du fuzzer : le projet SAGE

Avant d'aborder le fonctionnement du fuzzer, il convient de présenter succinctement l'algorithme utilisé. Tout comme Fuzzgrind, il repose sur le projet SAGE (*Scalable, Automated, Guided Execution*) de Microsoft Research [9]. SAGE est actuellement déployé sur environ 200 machines, et ses auteurs affirment qu'il a permis de trouver près d'un tiers des *bugs* de Windows Seven [10]. Le fonctionnement de cet algorithme est brièvement rappelé ci-dessous :

1. le binaire analysé est exécuté symboliquement avec une entrée initiale valide (fichier, entrée clavier, etc.) ;
2. à chaque branchement conditionnel, si celui-ci dépend de l'entrée initiale, la contrainte correspondante (c'est-à-dire l'ensemble des équations qui déterminent si le branchement est pris ou non) est enregistrée ;

3. chaque contrainte est alors inversée et envoyée à un solveur de contraintes. Si une solution existe, les valeurs des variables symboliques fournies par le solveur sont utilisées pour modifier l'entrée initiale ;
4. le binaire est exécuté « normalement » avec cette nouvelle entrée. Si elle ne provoque pas d'erreur, un score est donné à cette entrée, typiquement basé sur la couverture de code de l'exécutable ;
5. les nouvelles entrées sont triées par score décroissant, puis l'algorithme reprend à l'étape 1.

En comparaison avec les fuzzers utilisant des données aléatoires, l'algorithme se révèle très rentable : chaque itération va générer des nouvelles entrées « utiles », permettant à priori d'explorer une nouvelle branche d'exécution du programme. Cette technique impose cependant une gestion du nombre de contraintes qui peut rapidement exploser sur les programmes complexes.

### 1.2 Modules externes nécessaires

Pour implémenter cet algorithme dans le projet de portage, quatre principaux outils sont nécessaires :
- un solveur de contraintes (étape 3) ;
- un outil d'affectation de score aux nouvelles entrées (étape 4) ;
- un ordonnanceur qui gère la liste des fichiers à tester (étape 5) ;
- un outil d'instrumentation dynamique de binaire (*Dynamic Binary Instrumentation - DBI*), afin de pouvoir réaliser l'exécution concolique (étapes 1 et 2 de l'algorithme).

Concernant le solveur, Microsoft Z3 [19] a été choisi. Outre ses performances qui le place en tête des solveurs actuels, il est multi-plateforme (GNU/Linux, MacOS et Windows).

Particulièrement lié au projet SAGE, il dispose d'une documentation détaillée et était, au lancement de FuzzWin, l'un des seuls à utiliser la version 2 du langage SMT-LIB commun à tous les solveurs.

Si l'outil de couverture de code et d'ordonnancement des entrées sont aisément transposables au monde Windows, il n'en est pas de même pour le DBI. En effet, il n'existe aucune alternative à Valgrind qui offre une représentation intermédiaire indispensable à l'exécution concolique [1].

1. Une compatibilité de Valgrind sous Windows est offerte via le logiciel Wine, mais celle-ci est incomplète.

Une recherche rapide permet d'identifier deux principaux DBI sous Windows : DynamoRio et PIN. Leurs performances étant sensiblement équivalentes, le choix s'est porté sur PIN, en raison de ses nombreuses références dans le domaine du fuzzing, notamment aux conférences Black-Hat [6] et Hack In The Box [7]. Il dispose par ailleurs d'une API riche doublée d'une communauté importante et dynamique.

### 1.3 Architecture globale du projet FuzzWin

L'architecture de FuzzWin est résumée en figure 1. Le paragraphe suivant présente le cœur du projet de portage : la conception d'un langage intermédiaire spécialement développé pour ce projet.

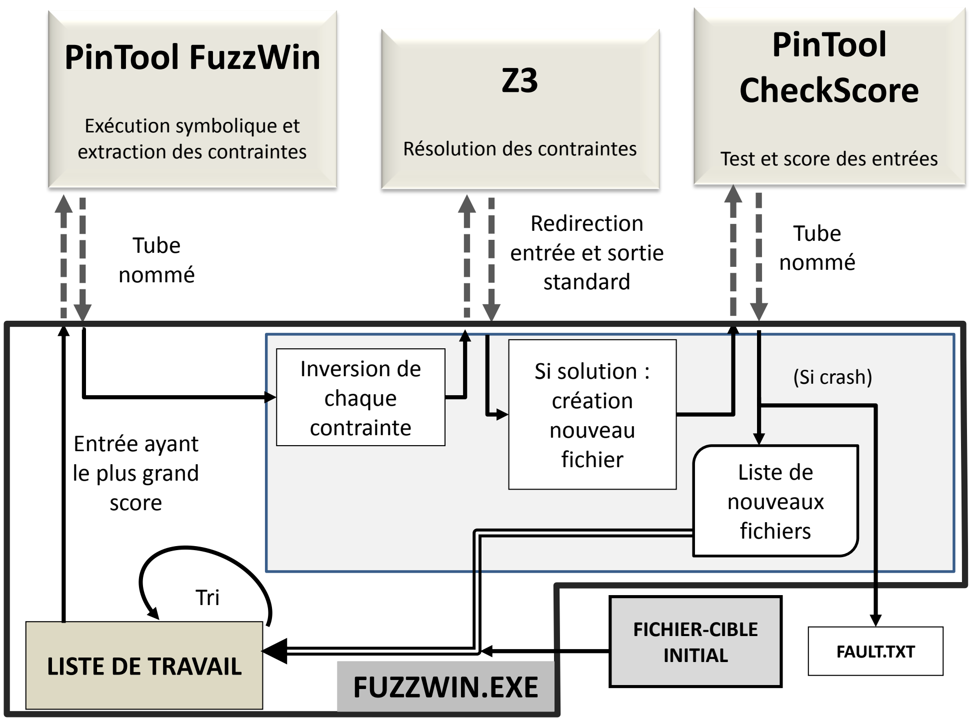


**Figure 1.** Architecture générale de FuzzWin

## 2 Exécution symbolique et représentation intermédiaire

### 2.1 Rappels sur l'exécution symbolique et concolique

L'exécution symbolique repose sur l'utilisation d'expressions algébriques pour représenter les valeurs des opérandes des instructions exécutées par un programme. Ainsi, l'instruction x86 *mov eax, ebx* peut se traduire, en langage symbolique, par la formule $a = b$, ou encore *AFFECT(a, b)* où $a$ et $b$ sont des variables de 32 bits représentant *eax* et *ebx*.

Lorsque l'exécution s'effectue dynamiquement, tout en prenant en compte les valeurs numériques réelles de la mémoire et des registres, l'expression « exécution concolique » est alors utilisée.

L'exécution concolique d'un binaire permet ainsi de transformer le code machine en une suite de formules compréhensible par un être humain ou, dans le cas du fuzzer dont il est question, par un solveur d'équations (ce qui n'est pas incompatible, fort heureusement). Ces formules constituent de fait une *représentation intermédiaire* du code exécuté, au sens ou la sémantique du code machine a été représenté dans un langage différent. Comme spécifié au paragraphe précédent, aucun DBI sous Windows n'offre à ce jour un tel niveau d'abstraction du code nativement.

## 2.2 Caractéristiques de la représentation intermédiaire

Plusieurs projets ont développé une représentation intermédiaire de code binaire, parmi lesquels *Miasm* [5], *Metasm* [11] ou encore le projet REIL [8]. Néanmoins, dans le cadre de FuzzWin, il a été décidé de développer une sémantique spécifique adaptée à l'emploi de PIN (et donc écrite en C++).

**Données nécessaires** Pour concevoir un langage intermédiaire modélisant une instruction écrite en code machine (ici x86, mais valable pour tout processeur), il est nécessaire de disposer de plusieurs données :
- la liste des opérandes *sources* de cette instruction ;
- la liste des actions unitaires de cette instruction sur les opérandes sources. En effet, il est rappelé que les processeurs x86 font partie de la famille des CISC [2] et que l'assembleur x86 peut être considéré parfois comme un langage de haut niveau ;
- la liste des opérandes de destination de cette instruction qui recevront le résultat des actions unitaires décrites *supra*.

Fort heureusement, pour le jeu d'instructions x86 ces données sont fournies par les deux principaux fabricants, Intel et AMD dans leurs manuels de développement [13,1], bien que certains effets soient partiellement documentés [3].

2. Complex Instruction Set Computer.
3. En particulier sur le registre *Eflags*.

**Choix des instructions à représenter** Dans le cadre de FuzzWin, il n'est pas nécessaire de procéder à l'abstraction de toutes les instructions rencontrées. En effet, l'algorithme du fuzzer repose sur le *marquage de données* : après avoir apposé une teinte sur les données de l'entrée initiale, FuzzWin suit la propagation de cette teinte tout au long de l'exécution du programme.

Dès lors, si aucune opérande source de l'instruction étudiée n'est marquée, alors aucune opérande de destination ne le sera : la représentation intermédiaire de cette instruction ne sera pas effectuée. Suivant la même logique, seules les opérandes sources qui sont marquées seront représentées par une variable symbolique.

Ce choix permet d'optimiser les performances de l'instrumentation du code tout en limitant le nombre de variables dans les formules symboliques. Ces dispositions sont résumées dans l'exemple décrit dans le tableau 1.

**Table 1.** représentation symbolique d'une addition dans FuzzWin

| eax | ebx | **add eax, ebx** |
|---|---|---|
| non marqué | non marqué | non traduit en langage intermédiaire |
| marqué($a$) | non marqué | addition ($a$, *Valeur de ebx*) |
| non marqué | marqué($b$) | addition(*Valeur de eax*, $b$) |
| marqué ($a$) | marqué ($b$) | addition($a$, $b$) |

## 2.3 Représentation du jeu d'instructions x86

Les caractéristiques du langage étant établies, il est possible de décrire les éléments constitutifs : la représentation des variables symboliques et la représentation des actions effectuées par l'instruction.

**Variables symboliques** Dans la représentation intermédiaire choisie pour FuzzWin, les variables symboliques sont constituées par des objets C++. Ce principe a été guidé par les travaux de Sean Heelan dans sa thèse soutenue à l'université d'Oxford [12].

Ces objets ont quatre caractéristiques principales :

- un entier représentant la taille de la variable, exprimée en bits ;
- un entier représentant la relation qui le lie à ses sources ;
- une liste d'« objets sources » ;
- une chaîne de caractères, le nom donné à la variable symbolique.

Les objets représentant les variables symboliques héritent de la classe *Taint* dont la déclaration est fournie au listing 1.

```
class Taint
{
  // taille de l'objet, en bits
  UINT32 _lenBits;
  // type de relation entre l'objet et ses sources
  const Relation _sourceRelation;
  // liste des sources de cet objet
  std::vector<ObjectSource> _sources;
  // nom de cet objet dans une formule au format SMT-LIB
  std::string _name;
  (...)
};

// patron de classe fille
template<UINT32 lengthInBits> class TaintObject : public Taint;
// alias pour les principales tailles
typedef TaintObject<1> TaintBit;  // objet de taille 1 bit (= Flag)
typedef TaintObject<8> TaintByte; // objet de taille 8 bits
typedef TaintObject<16> TaintWord;// objet de taille 16 bits
(...)
```

**Listing 1.** Variables symboliques : classe *Taint* et *TaintObject*

**Opérandes sources** En application du principe de marquage de données décrit au paragraphe précédent, seules les opérandes marquées sont représentées par une variable symbolique.

Le langage intermédiaire introduit donc également une classe *ObjectSource* permettant de représenter soit une variable symbolique, soit une valeur numérique. Ces objets sont ceux recensés dans le vecteur de sources décrit dans la classe *Taint*.

La définition de cette classe est précisée dans le listing 2. Il est à noter que cette représentation ne permet de traiter que les valeurs numériques d'une taille maximale de 64 bits (capacité maximale du type ADDRINT). Si l'instruction utilise des tailles supérieures [4], l'opérande source devra être représentée par plusieurs objets.

```
class ObjectSource
{
  // si la source est un objet : pointeur vers celui-ci, sinon NULL
  std::shared_ptr<Taint> _src;
  // sinon : sa valeur numerique et sa longueur en bits
  ADDRINT _val;
  UINT32  _lenBits;
```

4. C'est le cas de l'instruction CMPXCHG16B en architecture x64 par exemple.

```
public:
  // constructeur pour une source marquee
  ObjectSource(const std::shared_ptr<Taint> &tPtr)
      : _src(tPtr), _val(0), _lenBits(0) {}
  // constructeur pour une source numerique
  ObjectSource(UINT32 length, ADDRINT value)
      : _src(NULL), _val(value), _lenBits(length) {}
  (...)
};
```

**Listing 2.** Opérande source : classe *ObjectSource*

**Sémantique de l'instruction** La dernière caractéristique d'une variable symbolique est sa *relation*. Celle-ci est définie comme l'opération qui lie la variable à la liste de ses sources. Concrètement, elle est représentée par une simple énumération d'entiers, dont un extrait est fourni au listing 3.

```
enum Relation
{
    // variable issue de l'entree initiale. La source est une valeur
    // numerique representant l'offset de l'octet dans cette entree
    BYTESOURCE = 0,
    // construction d'objets par extraction ou
    // concatenation d'autres objets.
    EXTRACT,
    CONCAT,

    // affectation (= MOV); 1 seule source
    X_ASSIGN,

    // addition (2 sources)
    X_ADD,

    // rotation via Carry Flag (3 sources)
    X_RCL,

    // carry flag suite a addition (3 sources)
    F_CARRY_ADD,
    (...)
};
```

**Listing 3.** Énumération des relations entre une variable et ses sources

Les relations utilisées dans la représentation intermédiaire se divisent en trois grandes catégories :

– les relations qui représentent *l'action principale* d'une instruction x86 sur les opérandes sources. Ces relations ont le préfixe « X_ ». Les variables symboliques construites avec ce type de relation ont vocation à représenter la valeur d'une opérande de destination (registre ou mémoire) ;

– les relations qui représentent les *effets de bords* d'une instruction x86 sur le registre *Eflags*. Ces relations ont le préfixe « F_ ». Les variables symboliques construites avec ce type de relation sont toutes de la taille du bit, et représente la valeur d'un flag.
– les relations particulières, qui ne dépendent pas directement du jeu d'instruction x86. Cela concerne trois relations : la relation BYTESOURCE pour représenter un octet de l'entrée initiale (seule relation dont l'unique source est une valeur numérique), et celles qui permettent de construire des variables à partir d'objets de taille plus petite (CONCAT) ou plus grande (EXTRACT).

À ce jour le langage intermédiaire implémenté dans FuzzWin comporte 63 relations. Ce nombre peut paraitre important comparé aux autres langages connus [5]. Cependant, il est à noter que ces relations ne constituent qu'un « liant » entre les variables symboliques, et non pas la représentation intermédiaire définitive. Elles peuvent regrouper une ou plusieurs opérations unitaires, qui seront traduites dans un second temps selon les besoins de l'utilisateur.

Ainsi, la relation F_OVERFLOW_ADD, utilisée pour représenter le flag OF suite à une addition, pourra produire au final plusieurs formules :
– une simple expression intelligible par le commun des mortels :
$overflowFlag = valeurRetenueSuiteAdditionSignee(a, b, c)$ ;
– sa décomposition en opérations logiques de base :
$overflowFlag = [((\mathrm{a\,\hat{}\,b})\,\&\,(\mathrm{b\,\hat{}\,c})) >> (size\,(a) - 1)]\,\&1$ ;
– toute autre représentation imaginée et décrite par le programmeur

Dans le cadre de FuzzWin, les relations seront traduites en langage SMT-LIB V2 (cf. paragraphe 4).

### 2.4 Gestion du contexte d'exécution

Pour que la représentation intermédiaire soit fonctionnelle, l'utilisateur doit pouvoir récupérer la variable symbolique associée à chaque opérande source lors de l'exécution de l'instruction. Si l'opérande n'est pas marquée, sa valeur numérique sera fournie par le DBI lors de l'exécution.

À cet effet, FuzzWin introduit deux classes supplémentaires permettant de gérer, en temps réel, le contexte symbolique d'exécution du programme étudié :
– la gestion du marquage des registres généraux du processeur ainsi que le registre *Eflags* est effectuée au sein de la classe

5. À titre d'exemple, le langage REIL dispose d'un jeu réduit à 17 instructions.

*TaintManager_ Thread*. Chaque thread du programme étudié dispose de sa propre instanciation de cette classe ;
– la classe *TaintManager_ Global* est chargée de la gestion du marquage de la mémoire. Comme son nom l'indique, cette classe est commune à tous les threads du programme étudié.

Il est rappelé que FuzzWin ne s'intéresse qu'aux instructions et opérandes marquées. À cet effet, la convention suivante a été adoptée :
– si le registre ou l'emplacement mémoire est marqué, la classe de gestion contient un pointeur vers la variable symbolique qui le représente ;
– dans le cas contraire, le pointeur est nul, signifiant que l'emplacement concerné n'est pas marqué : une valeur numérique devra être employée.

Un extrait de l'implémentation de la classe *TaintManager_ Global* est présenté dans le listing 4. La classe de gestion du marquage des registres suit la même logique.

```
class TaintManager_Global {
private:
    // Marquage de la memoire : independante des threads
    std::map<ADDRINT, TaintBytePtr> _memoryPtrs;
public:
    // indique la plage [address, address+i] est marquee
    template<UINT32 i> bool isMemoryTainted(ADDRINT address) const;

    // renvoie un objet representant le marquage
    //  de la plage d'adresses
    template<UINT32 i> std::shared_ptr<TaintObject<i>>
    getMemoryTaint(ADDRINT address) const;

    // marquage de 'i' octets avec l'objet 'tPtr'
    // à partir de l'adresse 'address'
    template<UINT32 i> void updateMemoryTaint
    (ADDRINT address, const std::shared_ptr<TaintObject<i>> &tPtr);

    // Efface le marquage de la plage [address, address + i];
    template<UINT32 i> void unTaintMemory(ADDRINT address);
};
```

**Listing 4.** gestion du marquage de la mémoire

### 2.5 Granularité du marquage de données

La granularité du marquage de données est une problématique qui essaie de trouver le meilleur compromis entre finesse et performance. En effet, si le suivi au niveau du bit offre la meilleure authenticité, elle génère un coût non négligeable sur l'exécution au moment de l'instrumentation. *A contrario*, une vision trop macroscopique risque de générer un nombre conséquent de faux-positifs.

Partant du principe que FuzzWin suit la teinte issue d'une donnée initiale, et que celle-ci est acquise au niveau de l'octet (lecture dans un fichier ou au clavier), il a été décidé de choisir une granularité de 8 bits dans les classes de gestion, hormis le cas des flags qui sont, sans surprise, gérés au niveau du bit.

Ce choix explique la nécessite d'introduire les relations EXTRACT et CONCAT décrites précédemment. Celles-ci permettent de construire des variables symboliques « temporaires » de 16 bits ou supérieurs pour représenter les opérandes sources d'une instruction, par la concaténation de variables symboliques de 8 bits ou de valeurs numériques. La procédure est similaire pour le découpage en octets d'une opérande destination d'une instruction.

Ainsi, à titre d'exemple et pour synthétiser les processus décrits dans cette partie, FuzzWin effectuera les opérations suivantes pour l'instruction *add eax, ebx* :

1. si ni *eax*, ni *ebx* ne sont marqués : démarquer les *flags* qui seront forcément non marqués, et terminer le traitement ;
2. sinon, récupération de la variable symbolique associée à *eax*, qui sera la concaténation de 4 variables symboliques (ou valeurs numériques) de chaque octet du registre *eax*. Si aucun octet d'*eax* n'est marqué, récupération de sa valeur numérique sur 32 bits.
3. méthode similaire pour *ebx* ;
4. construction de l'objet *Taint* R, de taille 32 bits, comme la relation *X_ADD* entre les 2 objets *ObjectSource* construits à partir des variables symboliques (ou valeurs) définis ci-dessus ;
5. construction de 6 objets *Taint* OF, ZF, SF, AF, PF, CF, de taille 1 bit, représentant les effets de bords sur les *flags*. La relation et les objets *ObjectSource* dépendront de chaque *flag* ;
6. marquage des flags avec les objets OF, ZF, SF, AF, PF, CF.
7. marquage du registre *eax* avec la variable R. À ce titre, 4 objets R1, R2, R3 et R4 seront construits comme une extraction de R afin de marquer chaque octet du registre.

# 3 Utilisation de PIN pour l'instrumentation

## 3.1 Présentation de PIN

PIN est un logiciel d'instrumentation dynamique, développé par Intel. Bien que la licence soit propriétaire, la diffusion des outils utilisant PIN est autorisée pour peu que le code source soit fourni. PIN supporte les architectures x86 et x64, sur plateforme Windows et GNU/Linux. MacOS X et Android disposent d'une version expérimentale depuis début 2013 [16].

PIN est téléchargeable sous forme d'un kit adapté à la plateforme de développement [6]. Il fonctionne comme un *plugin* : partageant le même espace d'adressage que le programme instrumenté, il a accès à toutes les données de l'exécutable.

Dans son mode d'utilisation le plus courant, PIN agit comme un compilateur à la volée (JIT) : il recompile le code de l'exécutable étudié, en y insérant des fonctions d'analyse écrites par l'utilisateur, puis exécute ce code. L'insertion se fait au moyen de « hooks » sur des objets ou des évènements (exécution d'une instruction, chargement d'une DLL, nouveau *basic block*, changement de contexte, exception, appel système, etc.). Dans ce mode, les seules données réellement exécutées par l'OS sont celles générées par PIN. La figure 2 illustre ce fonctionnement.

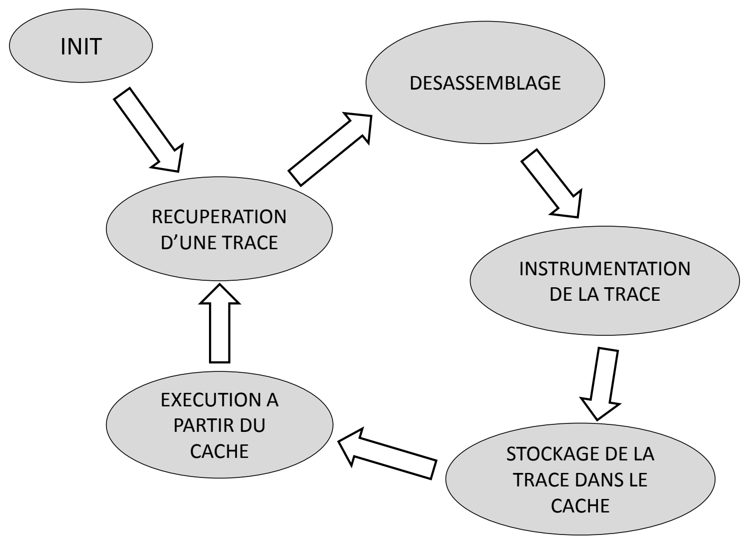


**Figure 2.** Schéma de fonctionnement de PIN en mode JIT

6. Il s'agit nécessairement de *Visual Studio* sous Windows.

On distingue trois grandes familles de fonctions :

- ***Les fonctions d'instrumentation***, appelées lors du désassemblage du programme original, avant la recompilation. Elles vont déterminer quel sera le code à injecter et son emplacement dans le code initial. L'instrumentation peut se faire à différentes granularités (instructions, *basic block*, trace, fonction, section, etc.) ;
- ***Les fonctions d'analyse***, qui constituent le code à injecter et qui sera exécuté par PIN. Elles peuvent recevoir de nombreux paramètres (valeur des registres, de la mémoire, etc.) qui sont déterminés lors de leur enregistrement par les fonctions d'instrumentation ;
- ***Les fonctions de notification***, qui constituent du code directement inséré lors de l'occurrence d'un évènement (appel système, démarrage et fin du programme, etc.). Elles sont de fait des fonctions d'analyse aux arguments prédéfinis.

Ces fonctions sont écrites en C++. Une fois compilées sous la forme d'une bibliothèque partagée, elles constituent un *pintool*.

### 3.2 Mise en place de l'instrumentation

Pour implémenter l'algorithme détaillé précédemment, FuzzWin utilise plusieurs fonctionnalités offertes par PIN :

**Instrumentation des appels systèmes** FuzzWin enregistre des fonctions de notification des appels systèmes. Ces notifications interviennent avant le passage en *ring0* (juste avant l'exécution de l'instruction *syscall* ou *sysenter*) et au retour en mode utilisateur. PIN n'est pas capable en effet d'instrumenter le code exécuté en mode *kernel*. Les appels systèmes suivis sont ceux qui manipulent les données du fichier d'entrée, à savoir *NtOpenFile/NtCreateFile*, *NtReadFile* et *NtOpenSection / NtMapViewOfSection*. Cette surveillance permet d'enclencher le marquage dès que les données du fichier d'entrée sont lues et stockées en mémoire.

Seul le numéro de l'appel système est fourni par PIN. Celui-ci est interprété, selon la version de Windows, grâce aux tables de correspondance existant sur Internet [14].

**Instrumentation des threads** Une fonction de notification est aussi enregistrée lors de la création d'un nouveau *thread* par le processus. Ce suivi permet de créer une nouvelle instance de la classe de gestion du marquage des registres dans la zone de stockage locale à chaque *thread* (ou TLS, *Thread Local Storage*), pour traiter le cas *multithreadé*. De même, FuzzWin suit la destruction des threads pour détruire la classe correspondante.

**Instrumentation des instructions** Enfin, des fonctions d'instrumentation et d'analyse de chaque instruction assembleur sont également mises en place, afin de suivre la propagation du marquage et surveiller les branchements conditionnels afin de détecter ceux qui sont marqués, et qui peuvent donc être influencés par la valeur des octets du fichier d'entrée. Ces fonctions ne sont pas insérées dès le lancement du programme, mais dès que les premières données ont été marquées (cf. paragraphes précédents). Afin d'optimiser les performances de PIN, les fonctions d'instrumentation vont récupérer le maximum d'information sur le code « statique ». Ainsi, les fonctions d'analyses seront spécifiques à chaque instruction et différenciées selon le type des opérandes source et destination (valeur, registre, mémoire) ainsi que sur la taille de ces opérandes.

Le listing 5 présente la simplicité d'implémentation de l'instrumentation dans la fonction *main*, et le listing 6 illustre un exemple d'enregistrement d'une fonction d'analyse. L'implémentation complète de la fonction d'analyse de l'instruction *XOR* entre une valeur numérique et un emplacement mémoire sur 8 bits est fourni dans le listing 10 en fin d'article.

```
#include "pin.h"

int main(int argc, char* argv[])
{
    // Initialisation de PIN
    if (PIN_Init(argc, argv)) PIN_ExitProcess(-1);

    // fonctions de notification des appels systemes
    PIN_AddSyscallEntryFunction(syscallEntry, 0);
    PIN_AddSyscallExitFunction(syscallExit, 0);

    // enregistrement d'une fonction d'instrumentation
    // appelee a chaque instruction rencontree
    INS_AddInstrumentFunction(Instruction, 0);

    // enregistrement d'une fonction de notification
    // appelee a la fin de l'execution du programme
    PIN_AddFiniFunction(Fini, 0);

    // Fonctions de notification de la creation et de la
    // suppression des threads de l'application
    PIN_AddThreadStartFunction(threadStart, 0);
    PIN_AddThreadFiniFunction (threadFini, 0);

    // Demarrage de l'instrumentation, ne retourne jamais
    PIN_StartProgram();
    return 0;
}
```

**Listing 5.** Fonction *main* du *pintool* FuzzWin

```
void Instruction(INS ins, void* v) {
    // si l'instruction est un ADD
    if (INS_Opcode(ins) == XED_ICLASS_ADD) {
        // si la source 1 est une valeur et la
        // source 2 est la memoire => cas ADD_IM
        if (INS_OperandIsImmediate(ins, 1) && INS_IsMemoryRead(ins))
        {
            // enregistrement de la fonction d'analyse de cette
            // instruction, avant son execution, avec 2 parametres
            INS_InsertCall (ins, IPOINT_BEFORE,
                // pointeur vers la fonction
                (AFUNPTR) callback_ADD_IM,
                // parametre 1 : source 1 (valeur)
                IARG_ADDRINT, INS_OperandImmediate(ins, 1),
                // parametre 2 : source 2 (adresse memoire)
                IARG_MEMORYREAD_EA,
                // fin des arguments
                IARG_END);
        }
        (...)
    }
}
```

**Listing 6.** Exemple d'enregistrement d'une fonction d'analyse

## 4 Application à l'algorithme de FuzzWin

Cette partie évoque l'utilisation du langage intermédiaire implémentée grâce au *pintool*, pour mettre en œuvre l'algorithme de Fuzzwin. Elle présente un exemple de traduction du langage défini aux paragraphes précédents pour une utilisation dans le cas du marquage de données.

### 4.1 Propagation du marquage et récupération des contraintes

Toutes les fonctions d'analyse suivent le même schéma : elles testent le marquage des opérandes sources, construisent le cas échéant les objets modélisant l'instruction étudiée et mettent à jour le marquage.

Plusieurs instructions ont un traitement particulier :

- **Les branchements conditionnels** du type *Jcc, CMOVcc* qui ont un comportement différent selon la valeur des drapeaux testés. Si ceux-ci sont marqués, la fonction d'analyse va construire une formule au format SMT-LIB, en introduisant une contrainte sur la valeur du drapeau ;
- **Les divisions** (*DIV / IDIV*). Dans le cas où le diviseur est marqué, une contrainte sur sa non-nullité sera enregistrée. Inversée, cette contrainte imposera une valeur nulle pour le diviseur. En cas de solution trouvée par le solveur, le crash de l'application est inévitable.

La figure 3 résume les procédés utilisés par le *pintool* de Fuzzwin.

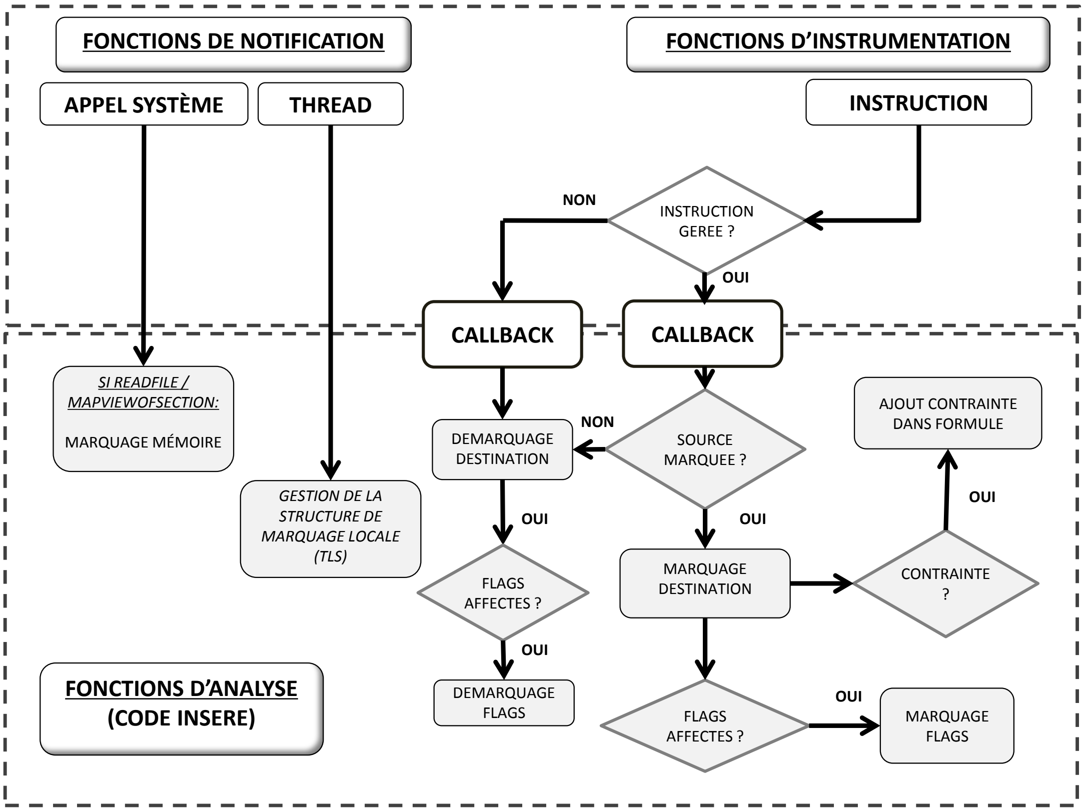


**Figure 3.** Fonctionnement de l'instrumentation au niveau du *pintool*

## 4.2 Traduction des contraintes en format SMT-LIB

**Brève description du langage SMT-LIB** SMT-LIB est une initiative internationale ayant pour objectif de faciliter la recherche dans le domaine des problèmes de décision de type SMT (*Satisfiability Modulo Theories*). SMT-LIB décrit en particulier une théorie relative aux vecteurs de bits de taille fixe (*fixed-size bitvectors*), qui modélise très bien l'arithmétique utilisée par les CPU de la famille x86. Par ailleurs, un langage commun est détaillé, tout comme la mise à disposition d'une importante base de données de tests afin de comparer les performances des différents solveurs SMT. Pour plus de précisions sur les aspects mathématiques sous-jacents, le lecteur pourra se référer à la page Wikipedia correspondante ainsi qu'au tutoriel disponible sur le site officiel de SMT-LIB [18,4].

**Emploi du langage SMT-LIB dans FuzzWin** Chaque contrainte marquée provoque la traduction des relations entre variables symboliques en une formule au format SMT-LIB. Les grands principes de cette traduction figurent ci-dessous (les références entre parenthèses renvoient aux exemples figurant dans le listing 7)

– le ou les objets impliqués dans le branchement vont être « déclarés », ainsi que leurs sources, et ainsi de suite récursivement, jusqu'à remonter vers les objets représentant les octets du fichier d'entrée. La déclaration consiste en l'affectation d'un nom de variable aux objets marqués, et la traduction en langage SMT-LIB de la *relation* entre l'objet et ses sources ;
– les noms de variable sont déterminés selon le type d'objet : pour un objet représentant un octet du fichier initial, le nom correspond à l'offset de l'octet dans le fichier *(1)(2)*. Les autres objets se voient affectés un nom et un numéro de variable en rapport avec la taille de l'objet (TBIT, TB, TW, TDW, TQW...). Enfin, les opérandes non marquées sont traduites par leurs valeurs numériques *(3)(5)* ;
– une contrainte SMT-LIB, ici sur un prédicat, s'exprime sous la forme d'une expression booléenne traduisant ce prédicat. Dans cet exemple, l'instruction *JNZ* teste si ZF est nul *(6)*. Ce flag a été traduit à la ligne précédente comme valant 1 si les deux opérandes de *(5)* sont égales, 0 dans le cas contraire. Enfin, la valeur de la contrainte lors de l'exécution est indiquée en fin de traduction *(6bis)* ; c'est cette valeur qui sera inversée dans la suite de l'algorithme.

La traduction est limitée aux seuls objets impliqués dans la contrainte. Par exemple, les instructions *(3)* et *(4)* provoquent le marquage d'autres *flags* du processeur, mais ceux-ci n'étant pas testés par *(6)*, aucune traduction des variables symboliques n'est réalisé.

```
movsx  eax, byte ptr [ebp-0xd9]  (1) (NB : memoire marquee)
movzx  edx, byte ptr [esi+ecx]   (2) (NB : memoire marquee)
add    eax, 0x45                 (3)
add    edx, eax                  (4)
cmp    edx, 0xffffff92           (5)
jnz    0x1381f53                 (6)

(declare-const OFF17 (_ BitVec 8))                              (1)
(define-fun TDW5 () (_ BitVec 32) ((_ sign_extend 24) OFF17)    (1)
(define-fun TDW6 () (_ BitVec 32) (bvadd TDW5 #x00000045))      (3)
(declare-const OFF14 (_ BitVec 8))                              (2)
(define-fun TDW7 () (_ BitVec 32) ((_ zero_extend 24) OFF14)    (2)
(define-fun TDW8 () (_ BitVec 32) (bvadd TDW6 TDW7))            (4)
(define-fun TBIT3 ()(_ BitVec 1)(ite(= TDW8 #xffffff92) #b1 #b0))(5)
(define-fun C_3() Bool (= TBIT3 #b0))   (6)
(assert (= C_3 false))                  (6bis)
```

**Listing 7.** Exemple de traduction en format SMT-LIB

### 4.3 Inversion et résolutions des contraintes

**Génération de nouveaux fichiers d'entrée** À la fin de l'instrumentation de l'exécutable, le *pintool* fournit un ensemble de formules SMT-LIB représentant les contraintes marquées rencontrées lors de l'exécution.

Grâce à la représentation intermédiaire spécifique, la formule est immédiatement « prête à l'emploi » : pour explorer les branches non prises, il suffit d'inverser la valeur du booléen de chaque ligne contenant l'instruction *assert* et de soumettre la formule au solveur. En cas de succès, les octets du fichier d'entrée sont modifiés avec les valeurs trouvées par le solveur.

Chaque formule fournie par le *pintool* permet de générer dans le meilleur des cas $N$ nouveaux fichiers dérivés, $N$ étant le nombre de contraintes présentes dans la formule.

**Rang d'inversion** L'algorithme utilisé dans SAGE introduit une optimisation relative au nombre de contraintes à inverser dans la formule générée.

À chaque fichier d'entrée est associée un entier, dénommé *bound*, qui représente un numéro de contrainte. Cet entier est utilisé comme suit :

- le fichier initial a un *bound* nul ;
- lors de l'inversion de la contrainte $n$ du fichier $F$, le fichier dérivé, noté *F'*, sera affecté d'un *bound* égal à $n$ ;
- l'exécution symbolique de *F'* génère une formule comprenant $N$ contraintes. Seuls les contraintes $n + 1$ à $N$ seront inversées l'une après l'autre. Si $N = n$, l'exécution de *F'* n'a pas pu générer de nouveaux chemins d'exécution : cette branche sera considérée comme « explorée ».

Cette méthode permet de n'inverser que les contraintes utiles au parcours de l'arbre d'exécution du programme. Elle évite en particulier que l'inversion de la contrainte $n$ de la formule issue de l'instrumentation de *F'* ne génère le fichier F.

**Utilisation du solveur dans FuzzWin** Bien que Z3 offre une API C++ pour permettre la construction de formules et leur résolution directement dans le code du *pintool*, cette opportunité n'a pas été utilisée, principalement afin de ne pas détériorer les performances. En effet, tout le code exécuté dans le *pintool* est générateur de surcoût en temps d'exécution.

Le solveur Z3 est exécuté en tant que processus « enfant » au sein du programme d'ordonnancement des fichiers de test.

Chaque nouvelle formule est traitée par le programme d'ordonnancement :

- les *bound* premières contraintes sont envoyées inchangées au solveur, puis la contrainte *bound + 1* inversée ; le solveur est alors interrogé, et les valeurs trouvées sont récupérées pour générer un fichier de *bound* égal à $n + 1$ ;
- la contrainte *bound + 1* originale est renvoyée, puis la contrainte *bound + 2* inversée est transmise. Le solveur est interrogé et ainsi de suite jusqu'à la dernière contrainte ;
- à la fin du traitement de la formule, le solveur est réinitialisé pour traiter les prochains fichiers.

## 4.4 Test et affectation du score

Les nouveaux fichiers générés sont immédiatement testés afin de vérifier s'ils provoquent une erreur dans l'application. Dans le cas contraire, un taux de couverture du code sera établi.

Ces actions sont réalisées par le *pintool* baptisé *CheckScore*. Ce *pintool* instrumente l'exécutable et détecte toute exception levée, tout en comptant le nombre d'instructions exécutées qui constituera le score du nouveau fichier.

Dans le cas d'une exception, l'exécutable sera alors lancé en mode *debug*, sans instrumentation, afin de récupérer l'adresse de l'instruction incriminée, et permettre une vérification *a posteriori* pour confirmer une éventuelle vulnérabilité exploitable.

Dans le cas contraire, le nouveau fichier est inséré dans une liste de test. Cette liste est triée par score décroissant à l'issue de la génération des fichiers dérivés du fichier *F*. Ainsi, le fichier au plus haut score est prioritairement exécuté.

Le listing 8 fournit l'implémentation volontairement simplifiée du *pintool CheckScore*.

```
static UINT64 count = 0;
static INT32 exceptionCode = 0;

static VOID insCount(TRACE trace, VOID *)
{
  BBL bbl = TRACE_BblHead(trace);
  while(BBL_Valid(bbl))
  {
    // ajout du nombre d'instructions de chaque basic block
    count += BBL_NumIns(bbl);
    bbl = BBL_Next(bbl);
  }
```

```
}

static void OnSig(THREADID tid, CONTEXT_CHANGE_REASON reason,
 const CONTEXT* cc, CONTEXT* c, INT32 sig, VOID* v)
{
  if (reason == CONTEXT_CHANGE_REASON_EXCEPTION)
  {
    // stockage du code d'exception fourni
    exceptionCode = sig;
    PIN_ExitApplication(-1);
  }
}

VOID Fini(INT32 code, VOID* v)
{
    if (exceptionCode)  /* Envoi du code d'erreur a l'utilisateur */
    else /* Envoi du nombre d'instructions a l'utilisateur */
}

int main(int argc, char *argv[])
{
    PIN_Init(argc, argv);

    PIN_AddContextChangeFunction(OnSig, 0);
    TRACE_AddInstrumentFunction(insCount, 0);
    PIN_AddFiniFunction(Fini, 0);

    PIN_StartProgram();
    return 0;
}
```

**Listing 8.** *pintool CheckScore*

## 5 Résultats obtenus par le fuzzer

Toujours en phase active de développement, FuzzWin est déjà pleinement fonctionnel sur Windows XP, Vista et Seven, en architecture 32 et 64 bits. Les tests de compatibilité avec Windows 8 sont en cours. Il ne supporte pour l'instant que les entrées de type fichier.

Afin d'optimiser le fonctionnement, il est possible de spécifier un temps maximal d'exécution de chaque itération, ou un nombre maximal de contraintes, ou encore une plage d'octets spécifique à marquer dans le fichier d'entrée.

### 5.1 Performances

Les expérimentations ont été effectuées sur une machine dotée d'un processeur Intel Core i5 cadencé à 2,6 Ghz, 8Gb de RAM, fonctionnant sous Windows 7 Professionnel 64 bits.

Les performances mesurées sont comparables à celles obtenues par Fuzzgrind sur GNU/Linux pour les programmes de test de type *strcmp* :

un fichier fautif est créé en moins de 25 secondes, après une dizaine de mutations du fichier initial.

Lors de tests sur des applications complexes de type lecteur PDF ou application bureautique, le surcoût en temps d'exécution engendré par l'instrumentation se fait ressentir : à titre d'exemple, *Adobe Acrobat* met plus de 50 secondes avant d'afficher le document fourni en argument. Les autres modules ont des performances tout à fait satisfaisantes : le délai de résolution des contraintes par le solveur est quasi-immédiat. Le *pintool* chargé du test et de la couverture de code engendre quant à lui un surcoût d'environ 30 % en temps pour l'exécutable.

## 5.2 Limitations et solutions envisagées

**Gestion des boucles** Les boucles sont la principale cause d'explosion du nombre de contraintes dans FuzzWin. Grâce aux outils d'instrumentation fournis par PIN, les instructions de manipulation des chaines de caractères (*STOS, LODS*...) sont maitrisées.

En revanche, dans les cas des boucles où le compteur est marqué, FuzzWin va générer autant de contraintes que d'itérations (cf. listing 9). L'une des solutions envisagées est de reconnaitre ces constructions soit en phase d'instrumentation, soit par étude de la formule SMT-LIB produite, afin de simplifier le traitement.

```
; le registre ecx est marque au debut de la boucle
; chaque instrumentation de l'instruction 'jnz loop' va provoquer
; l'enregistrement d'une contrainte

loop:
    mov ebx, dword ptr[esi + edx * 4]
    mov eax, dword ptr[edi + edx * 4]
    xor eax, ebx
    mov dword ptr[edi + edx * 4], eax
    inc edx
    dec ecx
    jnz loop
```

**Listing 9.** Exemple de code générateur de multiples contraintes

**Modélisation du marquage** Les opérandes de destination d'une instruction ne sont marquées que lorsque l'une au moins des opérandes source l'est. Cependant, il existe des cas où ce schéma ne s'applique pas si facilement, en particulier lors de la lecture en mémoire par un adressage indirect du type suivant :

```
mov eax, dword ptr[ebx + ecx * 4]
```

Il peut s'agir typiquement du résultat de la compilation du code C suivant :

```
int i;
uint32_t dest = tableau[i];
```

Si la valeur contenue dans le tableau à l'index *i* n'est pas marquée, alors *dest* ne sera pas marquée. Cependant, dans le cas où l'index *i* du tableau est marqué (ce qui signifie que *ebx* et/ou *ecx* sont marqués) la situation se complique sensiblement :

- selon la valeur de l'index, la zone de mémoire pointée, et donc la destination peut être marquée ou non ;
- il est difficile d'affirmer que la destination puisse ne pas être marquée, car sa valeur dépend justement de l'index du tableau !

Néanmoins, l'équation décrivant la valeur de la destination en fonction de l'index est quasi-impossible à écrire, sauf à faire un *dump* de la zone mémoire occupée par le tableau (indéterminé au niveau du code binaire), puis à fournir ces valeurs au solveur.

Ce type de problème a été soulevé de nombreuses fois dans les articles traitant du marquage de données :

- le projet *Privacy Scope* [20] a choisi de propager un marquage si le registre de base et le registre d'index sont marqués ;
- DTA++ [15] évoque la problématique, mais souligne que leurs tests sur les applications étudiées n'ont pas révélé de telles situations.

Dans le cadre de FuzzWin, il a été décidé que la destination serait, quoi qu'il arrive, démarquée. Des travaux sont en cours pour introduire une contrainte sur la valeur de l'adresse. Dans l'exemple précité, le solveur tentera de trouver des valeurs de *ebx* et *ecx* qui, additionnées, pointeront vers une autre adresse, donc un autre valeur du tableau (et potentiellement en dehors de ses bornes...).

**Temps d'exécution** Comme le souligne la documentation de PIN, l'instrumentation de l'application débute à la première instruction de l'exécutable fourni en argument. Or, sous Windows, le processus d'initialisation d'une application est long (chargement des DLL, création du contexte d'exécution. . .) et qui plus est sans intérêt pour le marquage des données. Bien que FuzzWin ne déclenche l'instrumentation des instructions que lorsque les premières données sont lues dans le fichier, cela affecte négativement les performances du fuzzing.

PIN offre une option pour être attaché à un processus existant, en lui fournissant son *pid*. Dès lors, il serait intéressant de pouvoir interfacer

FuzzWin avec un script d'automatisation tel AutoIt [7] ou Sikuli [8]. L'application serait alors démarrée par le script, attachée à PIN via son *pid* avant que le script ne provoque l'appel à la donnée initiale : ouverture du fichier, entrée clavier dans un formulaire. . .

## 6 Perspectives

À partir de la volonté initiale de portage d'un logiciel existant, le projet FuzzWin s'est rapidement transformé en un développement d'un langage intermédiaire pour des applications tournant sous Windows. Ce langage, et sa représentation dans la sémantique SMT-LIB, est construit à partir de données brutes fournies dynamiquement : la liste et la valeur des opérandes et le nom d'une instruction assembleur x86. À l'heure actuelle, le projet global représente plus de 15000 lignes de code C++.

La recherche de vulnérabilités au travers de l'exécution symbolique et du marquage de données est un domaine en pleine expansion. Pour preuve, les dizaines de références sur des projets en cours cités dans [7]. Ce paragraphe évoque brièvement les axes de développement futurs à la fois pour l'outil de fuzzing, mais aussi pour le langage intermédiaire.

**Portage vers d'autres OS** Les outils utilisés par FuzzWin (PIN et Z3) ont l'avantage d'être multi-plateformes. Par ailleurs, l'algorithme employé n'est pas dépendant de l'OS sous-jacent, hormis la surveillance des appels systèmes et les mécanismes de communication inter-processus (tubes nommés). Le domaine d'application de FuzzWin pourrait donc s'étendre vers les plateformes GNU/Linux, voire MacOS, et ce sans remettre en cause toute l'architecture du projet.

**Recherche d'autres types de vulnérabilités** L'objectif de FuzzWin est de parcourir idéalement l'ensemble des branches d'un exécutable, et de provoquer une faute de ce dernier avec une entrée calculée. L'exécution concolique permet également de rechercher des vulnérabilités dans les programmes de type *Buffer Overflow* ou *Use After Free*.

Beaucoup d'écrits existent sur ce sujet, en particulier les travaux de Johnatan Salwan [9]. Son blog détaille plusieurs approches permettant la détection de vulnérabilités, grâce notamment à l'instrumentation de fonctions spécifiques (*malloc, free. . .*) ou de la surveillance de la pile.

7. `http://www.autoitscript.com/site/`
8. `http://www.sikuli.org/`
9. `www.shell-storm.org`

Ces concepts pourraient enrichir l'instrumentation mise en place par FuzzWin, en fournissant une approche non plus basée sur la recherche d'autres chemins d'exécution mais sur la levée d'alertes en cas de suspicion de problème dans le code. Ces alertes seraient fournies en fin de fuzzing pour contrôler leur pertinence par un examen manuel.

**Enrichissement du langage** L'implémentation actuelle du langage intermédiaire adresse l'ensemble des instructions x86 qui traitent des opérandes entières, soit un peu plus d'une centaine.

La conception du langage intermédiaire permet d'ajouter très facilement une nouvelle instruction en suivant la méthode décrite précédemment :

- utilisation d'une relation existante ou introduction d'une nouvelle relation pour décrire le fonctionnement de l'instruction ;
- enregistrement d'une fonction d'analyse dans le *pintool*, avec les arguments *adhoc* ;
- écriture de la fonction d'analyse correspondante ;
- écriture de la représentation intermédiaire de la relation dans un langage de sortie (SMT-LIB ou autre).

Ainsi, les instructions SIMD sur les entiers (MMX, SSE, AVX) seront très prochainement ajoutées : leur fonctionnement est analogue aux instructions déjà gérées. La principale différence réside dans la taille des opérandes.

L'ajout des instructions sur les flottants n'est pas plus complexe à réaliser. En effet, le langage SMT-LIB dispose d'une logique spécifique aux nombre réels, et il est ainsi possible de modéliser l'effet de ces instructions. L'implémentation n'a pas été réalisée pour l'instant, car le mélange entre entiers et flottants au sein d'une même session est complexe à réaliser.

**Utilisation du langage dans d'autres projets** Le langage intermédiaire mis en œuvre dans FuzzWin a été spécialement pensé pour traiter le marquage de données. Néanmoins, rien n'interdit son emploi pour d'autres objectifs :

- récupération de la fonction de transfert d'un *basic block* ;
- aide à la désobfuscation de code grâce à la fonction de simplification des formules SMT-LIB offerte par le solveur Z3 ;
- adaptation du langage pour pouvoir servir dans des *plugins* d'outils tels les debuggers, etc.
- utiliser un autre DBI que PIN pour pouvoir prendre en compte l'instrumentation des instructions en mode *kernel*.

## 7 Conclusion

Si de nombreuses articles universitaires évoquent les théories décrites dans cet article, à la connaissance de l'auteur peu de projets s'aventurent cependant dans l'implémentation d'un *framework* complet, qui plus est sur les plateformes de type Windows. Lorsque cela est le cas, comme dans BitBlaze [17,2], certaines parties du code sont diffusées (module *Vine* par exemple), mais le module traitant spécifiquement de l'exécution symbolique (*Rudder*) n'est pas disponible.

Toujours en phase de développement actif, FuzzWin offre un langage intermédiaire abstrait et un premier dictionnaire de traduction en langage SMT-LIB, le tout sur plateforme Windows. Son objectif premier est de réaliser le fuzzing intelligent de binaires afin de trouver des vulnérabilités. Cependant l'emploi du langage intermédiaire peut ouvrir la voie à de nombreuses autres applications. Le code source de FuzzWin est public et disponible librement (`https://github.com/piscou/FuzzWin/`). À ce titre, toutes les remarques, les encouragements, les propositions voire les contributions sont les bienvenus !

```
template void LOGICAL::sXOR_IM_<8>(
THREADID tid, ADDRINT value, ADDRINT writeAddress) {
    TaintManager_Thread *pTmgrTls = getTmgrInTls(tid);
    if ( !pTmgrGlobal->isMemoryTainted<8>(writeAddress)) {
        pTmgrTls->unTaintAllFlags(); // operande source non marquee
    }
    else {
        // cas 1 : si value vaut 0, alors XOR x, 0 = x
        // donc faire juste marquage des flags avec destination
        if (!value) {
            fTaintLOGICAL(pTmgrTls, pTmgrGlobal->getMemoryTaint<8>(
                writeAddress));
        }
        // cas 2 : si value vaut 0xff  alors XOR x, 0xff = NOT x
        else if (value == 0xff) {
            TaintBytePtr rPtr = std::make_shared<TaintByte>(X_NOT,
                ObjectSource(pTmgrGlobal->getMemoryTaint<8>(
                    writeAddress)));

            // marquage des flags + dest
            fTaintLOGICAL(pTmgrTls, rPtr);
            pTmgrGlobal->updateMemoryTaint<8>(writeAddress, rPtr);
        }
        // cas 3 (general) : XOR "normal"
        else {
            TaintBytePtr rPtr = std::make_shared<TaintByte>(X_XOR,
                ObjectSource(pTmgrGlobal->getMemoryTaint<8>(
                    writeAddress)),
                ObjectSource(8, value));

            // marquage des flags et de la destination
```

```
            fTaintLOGICAL(pTmgrTls, rPtr);
            pTmgrGlobal->updateMemoryTaint<8>(writeAddress, rPtr);
        }
    }
} // sXOR_IM_8BITS

void LOGICAL::fTaintLOGICAL(TaintManager_Thread *pTmgrTls,
    const TaintPtr &resultPtr) {
    // -> OF et CF mis a 0 donc demarquage
    pTmgrTls->unTaintCarryFlag();
    pTmgrTls->unTaintOverflowFlag();

    // -> AF 'undefined' selon Intel donc demarquge
    pTmgrTls->unTaintAuxiliaryFlag();

    //  SF/PF/ZF marquage avec les relations adequates
    ObjectSource objResult(resultPtr);
    pTmgrTls->updateTaintParityFlag(
        std::make_shared<TaintBit>(F_PARITY, objResult));
    pTmgrTls->updateTaintZeroFlag(
        std::make_shared<TaintBit>(F_IS_NULL, objResult));
    pTmgrTls->updateTaintSignFlag(
        std::make_shared<TaintBit>(F_MSB, objResult));
} // fTaintLOGICAL
```

**Listing 10.** Exemple de fonction d'analyse : XOR 8 bits IM

## Références


1. *AMD64 Architecture Programmer's Manual Volume 3*. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/.
2. BitBlaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu/.
3. Gabriel Campana. Fuzzgrind : un outil de fuzzing automatique. In *Actes du SSTIC 2009*, 2009. https://www.sstic.org/2009/presentation/Fuzzgrind_un_outil_de_fuzzing_automatique/. Voir également la page du projet : http://esec-lab.sogeti.com/pages/Fuzzgrind.
4. David R. Cok. The SMT-LIB V2 Language and Tools : A Tutorial - Version 1.2.1. http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf.
5. Fabrice Desclaux. Miasm, un framework de reverse engineering open-source. https://code.google.com/p/miasm.
6. Gal Diskin. Binary instrumentation for security professionnels. In *USA Blackhat briefings & training*, 2011. http://media.blackhat.com/bh-us-11/Diskin/BH_US_11_Diskin_Binary_Instrumentation_Slides.pdf.
7. Gal Diskin. Hacking using dynamic binary instrumentation. In *Hack In the Box*, Amsterdam, 2012. http://conference.hitb.org/hitbsecconf2012ams/materials/.
8. Thomas Dullien and Sebastian Porst. Reil : A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*, 2009. http://blog.zynamics.com/2010/08/24/the-reil-language-part-iv/.

9. Patrice Godefroid, Michael Levin, Y., and David Molnar. Automated whitebox fuzz testing. In *NDSS 2008, 15th Annual Network & Distributed System Security Symposium*, pages 151–166, San Diego, 2008. http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ndss2008.pdf.
10. Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage : Whitebox fuzzing for security testing. In *Communications of the ACM*, volume 55-3, pages 40–44, March 2012. http://research.microsoft.com/en-us/um/people/pg/public_psfiles/cacm2012.pdf.
11. Yoann Guillot and Juilen Tinnes. The metasm assembly manipulation suite. http://metasm.cr0.org/.
12. Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Msc computer science dissertation, University of Oxford - Computing Laboratory, September 2009. http://www.cprover.org/dissertations/thesis-Heelan.pdf.
13. *Intel® 64 and IA-32 Architectures Software Developer Manuals Vol 2 : Instruction Set Reference.* http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
14. Mateusz "j00ru" Jurczyk. Windows x86 system call table. http://j00ru.vexillium.org/ntapi/.
15. Min G. Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Ong. DTA++ : Dynamic taint analysis with targeted control-flow propagation. In *NDSS 2011, 18th Annual Network & Distributed System Security Symposium*, Washington, DC, USA, 2011. http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_4.pdf.
16. Page officielle de PIN. http://software.intel.com/en-us/articles/pintool.
17. 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 *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.
18. Page Wikipedia consacrée aux théories SMT. http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories.
19. Page officielle de z3. http://z3.codeplex.com/.
20. Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Privacy scope : A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, University of California, Berkeley, 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-145.html.