Fuzzgrind : un outil de fuzzing automatique

Gabriel Campana

Sogeti/ESEC

Résumé Le fuzzing est devenu au cours de ces dernières années une des principales méthodes de recherche de vulnérabilités. Les fuzzers actuels reposent principalement sur la génération de données invalides à partir de modèles, qui nécessitent d'être réécrit pour chaque nouvelle cible. Nous détaillerons le fonctionnement de *Fuzzgrind*, un nouvel outil de fuzzing entièrement automatique, générant de nouvelles données de test à partir de l'exécution symbolique du programme cible sur une entrée donnée, dans le but de rechercher de nouvelles vulnérabilités.

1 Introduction

1.1 Définition du fuzzing et outils actuels

Le *fuzzing* est une technique de recherche d'erreurs d'implémentation logicielle par injection de données invalides. L'injection de données invalides est bien entendu relative à l'implémentation logicielle testée, pour trouver les tests les plus pertinents susceptibles de découvrir des erreurs d'implémentation classiques (débordement de tampon, bug de chaîne de format, etc).

Il existe actuellement une multitude de logiciels de fuzzing [1], générant principalement les données de tests de deux façons :

- aléatoirement,
- à partir d'une grammaire ou d'un modèle auquel certaines heuristiques de mutation sont appliquées pour rendre les données générées invalides.

Au vu des vulnérabilités trouvées par cette seconde catégorie de fuzzer, leur efficacité et leur puissance n'est plus à prouver. Ils possèdent cependant de nombreux inconvénients et limitations.
L'écriture du modèle ou de la grammaire est un processus sans fin, complexe et fastidieux, nécessitant de disposer des spécifications du protocole ou du format de fichier cible (ce qui n'est bien évidemment pas toujours le cas, en particulier pour les logiciels propriétaires). Les étapes d'ingénierie inverse, d'étude des spécifications et de description de la grammaire prennent un temps conséquent, et doivent de plus être renouvelées pour chaque nouveau protocole ou format de fichier testé.
L'implémentation cible, quant à elle, ne respecte pas nécessairement les spécifications.

[1] Liste de frameworks et de logiciels de fuzzing : http://www.nosec.org/web/fuzzers.

Bien que fortement susceptibles de comporter des vulnérabilités, les fonctionnalités non documentées n'auront que peu de chance d'être testées.
Enfin, certaines vulnérabilités semblent peu enclines à être trouvées par ce moyen. Cet exemple de buffer overflow dans la commande PASS d'un serveur FTP, seulement déclenché lorsque le mot de passe commence par une virgule [2], illustre parfaitement le fait que de nombreuses vulnérabilités échapperont toujours à cette classe de fuzzers.

Parallèlement, de nouveaux fuzzers reposant sur des théories innovantes ont vu le jour :

- Autodafé [1] analyse tous les appels de fonctions « dangereuses » d'un programme pendant son exécution, à l'aide du debugger *gdb*. Si ces fonctions dépendent de données contrôlées par l'utilisateur, elles sont mutées dans le but de déclencher un *buffer overflow*.
- Catchconv [2] est un outil de génération de données destinées à déclencher des *integer overflows*, basé sur *Valgrind* et *STP*.
- Bunny the Fuzzer [3] injecte dans la source du programme cible des fonctions d'instrumentation pendant la compilation. Ces fonctions permettent au fuzzer de recevoir des informations sur le chemin d'exécution, les arguments des fonctions, les valeurs de retour, ainsi que de modifier ces valeurs à l'exécution.
- Flayer [4] est un plugin pour Valgrind reposant sur la technique de *tainted data flow analysis*, destiné à tracer et modifier manuellement le chemin des entrées. L'approche utilisée par Fuzzgrind est une extension audacieuse de celle utilisée par Flayer.
- DART [5], CUTE [6], et EXE [7] extraient du code source les prototypes des fonctions, et génèrent aléatoirement des entrées. Le programme cible est exécuté symboliquement sur ces entrées pour inverser chaque *path condition*.
- SAGE [8] émule et trace symboliquement les instructions du programme cible afin de générer des données de test pour fuzzer des formats de fichier sous Windows.

L'existence de ces fuzzers montre qu'il n'y a pas une seule et unique technique de fuzzing, et que toutes les voies de recherche sont loin d'être explorées. Les vulnérabilités trouvées par ces méthodes sont la plupart du temps différentes ; chacune d'entre elle est spécifique à une cible particulière : protocoles réseaux, formats de fichier, langage de scripts, etc.

[2] http://blogs.securiteam.com/index.php/archives/677

1.2 Concept de Fuzzgrind

L'exécution symbolique, un concept introduit par King dans les années 1970, fut le point de départ de plusieurs projets de recherche d'erreurs dans des programmes, notamment [2,5,6,7,8] qui ont inspiré le fonctionnement de Fuzzgrind. L'exécution symbolique d'un programme consiste à utiliser des expressions algébriques pour représenter les valeurs des variables tout au long de l'exécution. Le chemin d'exécution du programme (c'est à dire la suite de branches conditionnelles prises ou non par le programme) est représenté par les *path conditions*, qui correspondent aux contraintes sur les variables symboliques devant être satisfaites pour exécuter le chemin. Les résultats de l'exécution peuvent ainsi être représentés à l'aide des valeurs symboliques des variables et des conditions de chemins ; comme le montre la figure 1.

```
//      execution concrete   |      execution symbolique
//      ---------------------+---------------+-------------
//            etat concret   | etat symb.    | path constr.
//      ---------------------+---------------+-------------
int calc(uint y) {   // y=4  | 0 <= y < 2^32|
  uint x = 2 * y;    // x=8  | x = 2y        |
                     //      |               |
  if (y % 2 == 0) {  //      |               | y % 2 == 0
    if (y > 10) {    //      |               | y > 10
      y += x;        //      |               |
    }                //      |               |
    y *= 5;          // y=20 | y = 5y        |
  }                  //      |               |
                     //      |               |
  return y;          //      |               |
}
```

Fig. 1. Représentation des exécutions concrètes et symboliques d'un programme.

L'approche utilisée par Fuzzgrind est d'analyser les conditions de chemin de l'exécution symbolique d'un programme sur une entrée donnée afin de générer de nouvelles entrées, susceptibles de provoquer des erreurs (et pouvant potentiellement être la source de vulnérabilités). Le programme cible est exécuté symboliquement sur une entrée donnée (typiquement un fichier ou l'entrée standard), enregistrant toutes les contraintes symboliques liées aux instructions de branchement conditionnelles dépendant de l'entrée rencontrée sur le chemin d'exécution. Chaque contrainte est inversée pour que la branche qui n'a pas été suivie le soit. Un solveur de contraintes est appelé sur ces nouvelles contraintes pour générer de nouvelles entrées de test.

L'exécution suivante du programme sur ces nouvelles entrées suivra ainsi de nouveaux chemins d'exécution. Cet algorithme est répété (en suivant une heuristique basée sur la couverture de code pour exécuter en priorité le programme sur des entrées susceptibles de déclencher des erreurs), idéalement jusqu'à ce que tous les chemins d'exécution soient couverts.

Ces idées ont été implémentées dans *Fuzzgrind*. Nous détaillerons dans la suite de cet article son fonctionnement ainsi que celui des outils utilisés sur lesquels repose notre implémentation, *Valgrind* et *STP*. Enfin, nous analyserons les résultats obtenus sur des exemples simples, et des librairies utilisées par des programmes répandus, ainsi que les améliorations possibles.

2 Description des logiciels Valgrind et STP

Fuzzgrind repose sur les deux logiciels libres Valgrind [3] et STP [4]. Valgrind est un framework d'instrumentation binaire dynamique (Dynamic Binary Instrumentation) sous licence GPL, supportant de nombreuses architectures [5] et destiné à la création d'outil d'analyse binaire dynamique (Dynamic Binary Analysis). STP est un solveur de contraintes sous licence MIT. Seuls les mécanismes internes de Valgrind et STP nécessaires à la compréhension de Fuzzgrind seront exposés dans la suite de cette partie. L'article [9] présente en détail les fonctionnements complets de Valgrind et Memcheck. Le fonctionnement interne et les théories derrière STP sont détaillés dans l'article [10].

2.1 Valgrind

Présentation générale Le framework d'instrumentation binaire dynamique Valgrind comporte par défaut quatre outils dont le célèbre Memcheck [6], destiné à découvrir des problèmes d'utilisation de la mémoire. Son architecture est modulaire, de façon à ce que de nouveaux outils puissent être créés facilement sans avoir à modifier sa structure interne. Le cœur de Valgrind est constitué de deux parties, *coregrind* et *VEX*. VEX est responsable de la traduction dynamique de code et de l'appel des fonctions de l'outil pour l'instrumentation de la représentation intermédiaire, tandis que coregrind est responsable de tout le reste (l'ordonnancement, le cache des blocs, la gestion des symboles, etc).

[3] http://www.valgrind.org
[4] http://people.csail.mit.edu/vganesh/STP_files/stp.html
[5] x86, x86-64, PPC32 et PPC64 sous le système d'exploitation GNU/Linux ; PPC32 et PPC64 sous AIX
[6] http://valgrind.org/docs/manual/mc-manual.html

Coregrind Après avoir lancé Valgrind en spécifiant un outil et un programme cible, le cœur, l'outil et le programme cible sont chargés dans un même processus. Le code machine du programme cible est analysé par *basic block* [7] de cinquante instructions maximum.

Valgrind ne peut évidemment pas analyser le code exécuté par le noyau. Lorsqu'un appel système est effectué par le programme cible, le contrôle revient à l'ordonnanceur, qui effectue quatre opérations :

- tous les registres du programme cible sont copiés à la place du programme hôte, hormis le pointeur d'instruction (Program Counter, PC),
- l'appel système est appelé,
- les registres de l'invité sont copiés, hormis le pointeur d'instruction,
- le pointeur de pile de l'outil est restauré.

Les appels système nécessitant des ressources partagées entre les deux processus comme de la mémoire (par exemple *mmap*) ou des descripteurs de fichier (par exemple *open*) sont vérifiés pour s'assurer qu'ils ne causent pas de conflit avec l'outil. Par exemple, si le client tente de *mmaper* de la mémoire utilisée par l'outil, Valgrind le fait échouer sans effectuer aucun appel système.

VEX La traduction des blocs est effectuée par VEX en huit phases consécutives :

1. Le désassemblage du code du programme cible : le code dépendant de l'architecture sur laquelle le programme est lancé, est traduit dans la représentation intermédiaire de VEX, indépendant de l'architecture.
2. L'optimisation de la représentation intermédiaire.
3. L'instrumentation : VEX appelle la fonction principale de l'outil sélectionné, responsable de l'instrumentation de la représentation intermédiaire.
4. L'optimisation de la représentation intermédiaire, similaire à la phase 2.
5. *Tree building* : mise à plat de la représentation intermédiaire pour simplifier la phase 6.
6. La sélection d'instructions : la représentation intermédiaire est convertie en code machine.
7. L'allocation des registres : les registres réels de la machine hôte sont alloués.

[7] Bloc d'instructions sans aucun *jump* (instruction de branchement) ni destination de jump ; la destination d'un jump désigne le début du bloc, et un jump la fin).

8. La génération du code final : le code machine final est généré, en encodant les instructions générées précédemment et en les enregistrant dans un bloc de mémoire.

À la fin de chaque bloc, VEX sauve tous les registres invités en mémoire, pour que la traduction de chaque bloc soit indépendante des autres. Nous noterons que Valgrind recompile chaque instruction du code machine du programme cible à la volée, et qu'à aucun moment, le code original du programme est exécuté.

L'instrumentation d'un programme par Valgrind possède néanmoins un impact important sur les performances. Avec un plugin n'effectuant aucune action, l'exécution du programme cible par Valgrind prend quatre fois plus de temps. Avec le plugin Memcheck, la taille du code exécuté est multipliée par douze, et l'exécution est vingt-cinq à cinquante fois plus lente.

Représentation intermédiaire et instrumentation Le code d'instrumentation du plugin Fuzzgrind est appelé par VEX pendant la troisième phase. La représentation intermédiaire de VEX est consituée de petits blocs de cinquante instructions maximum. Une instructions du code machine original est traduite en une ou plusieurs instructions de la représentation intermédiaire. Celle-ci est constituée d'instructions avec effet de bord (enregister une valeur en mémoire, assigner une valeur à un temporaire, etc), manipulant des expressions sans effet de bord (opérations arithmétiques, chargement d'une valeur enregistrée en mémoire, etc). Une traduction de code machine en représentation intermédiaire est présentée par la figure 2.

```
0x4000A99:  movl %eax,%ecx
  ------ IMark(0x4000A99, 2) ------
  PUT(4) = GET:I32(0)               ; copie d'EAX dans ECX

0x4000A9B:  leal 0x2C(%ebx), %esi
  ------ IMark(0x4000A9B, 6) ------
  PUT(60) = 0x4000A9B:I32           ; mise à jour d'EIP
  t0 = Add32(GET:I32(12),0x2C:I32)  ; addition du contenu d'EBX avec 0x2C
  PUT(24) = t0                      ; copie du érsultat dans ESI
```

Fig. 2. Exemple de code machine x86 désassemblé en représentation intermédiaire, pendant la phase 1.

Le nombre de registres est fixe, tandis que le nombre de temporaires n'est pas limité. Un environnement de types contient la taille de chaque temporaire. L'ensemble de la représentation intermédiaire est décrit dans `VEX/pub/libvex_ir.h`, et expliqué plus en détail dans [2]. Chaque *basic block* est instrumenté séparément dans Valgrind. Les outils implémentent une fonction de callback qui prend en paramètre un basic block, et retourne un pointeur vers une copie du basic block instrumenté. Cette fonction peut boucler sur chaque instruction du basic block en entrée, et le copier normalement dans le basic block de sortie. L'instrumenteur peut ajouter, supprimer, ou modifier des instructions. Il est par exemple possible d'ajouter directement des appels de fonction dont le corps peut être écrit en C dans le code source de l'outil.

2.2 STP

Présentation du langage STP est un solveur de contraintes générées en pratique par des programmes d'analyse statique et dynamique. Il est notamment utilisé dans le projet de génération automatique de *proof of concepts* à partir de patches décrit par l'article [11], et dans Catchconv.

STP accepte en entrée une unique requête constituée d'une ou plusieurs contraintes. La sortie indique si la formule est satisfiable ou non, et le cas échéant, une valeur est assignée à chaque variable pour exhiber un contre-exemple. Le langage de STP [8], constitué d'un ensemble de fonctions et de prédicats, est suffisant pour représenter les contraintes correspondant aux conditions d'un programme. Ce langage est caractérisé par :

- des fonctions de manipulation de variables : `CONCAT`, `EXTRACT`, `>>`, `<<`, `SIGN EXTEND`.
- des fonctions de logique combinatoire : `AND`, `OR`, `NOT`, `XOR`, `NAND`, `NOR`, `XNOR`.
- des fonctions arithmétiques : `ADD`, `MULT`, `SUB`, `DIV`, `SIGNED DIV`, `MODULO`, `SIGNED MODULO`.
- des prédicats : `=`, `<`, `>`, `<=`, `>=` signés et non signés.

La figure 3 présente l'exécution de STP sur une contrainte. Deux variables x et y de 8 bits sont déclarées. La formule requiert que le résultat de la multiplication sur 8 bits de x et y soit différent de 16. Cette contrainte n'est pas satisfiable puisqu'il existe au moins un couple de valeurs (x, y) tel que $(x * y) \% 256 = 16$, comme le montre le contre-exemple résultant de l'exécution de STP. La contrainte est donc

[8] décrit succinctement dans la documentation accessible à l'adresse http://people.csail.mit.edu/vganesh/STP_files/stp-docs.html.

invalide avec les valeurs $x = 5$ et $y = 208$.

```
% cat file.stp
x : BITVECTOR(8);
y : BITVECTOR(8);
QUERY(NOT(BVMULT(8, x, y) = 0h10));

% stp -p file.stp
Invalid.
ASSERT( y  = 0hex05  );
ASSERT( x  = 0hexD0  );
```

Fig. 3. Exemple de résultat d'exécution de STP sur une contrainte.

3 Fuzzgrind

Le but de Fuzzgrind est de trouver des vulnérabilités dans des programmes en partant d'une entrée fixée, et en générant de nouveaux fichiers de test pour passer par de nouveaux chemins d'exécution. Fuzzgrind fonctionne directement sur le code machine du programme cible, et n'a pas besoin de ses sources.

3.1 Algorithme

L'algorithme utilisé est celui implémenté dans SAGE et décrit par Microsoft dans [8]. Il est divisé en deux parties : la fonction `search` définie dans la figure 4 prend en argument le fichier ou l'entrée standard initiale, `input_seed`. La liste `worklist`, initialisée avec `input_seed`, sera étendue par les nouveaux fichiers générés. Le premier élément est retiré de la liste et développé par la fonction `expand_execution`, qui génère de nouveaux fichiers. Une note est attribuée à chacun de ces nouveaux fichiers par la fonction `score`, pour exécuter en priorité les fichiers dont la couverture de code est maximale. L'algorithme est répété sur l'entrée possédant le score le plus élevé tant que la liste `worklist` n'est pas vide.

La fonction `expand_execution` de la figure 5 est destinée à générer de nouvelles entrées, à partir de l'entrée `input` donnée en paramètre. Une liste `pc` de conditions de chemin est générée par l'exécution symbolique du programme cible sur l'entrée `input`. Après avoir été optimisées, ces conditions sont inversées une à une, et passées à un solveur de contraintes par la fonction `solve`. S'il existe une solution à ces conditions,

```
def search(input_seed):
    worklist = [ input_seed ]

    while worklist:
        # entree possedant le score le plus eleve
        input = worklist.pop()

        # execution symbolique, resolution des
        # contraintes, et creation de nouveaux
        # fichiers de test
        child_inputs = expand_execution(input)

        for input in child_inputs:
            input.fault = check(input)

        # ajout des nouveaux fichiers de tests
        worklist.append(child_inputs)

        # tri de la liste en fonction du resultat de
        # la fonction score appliquee a chaque fichier
        worklist.sort(score)
```

Fig. 4. Algorithme d'exécution symbolique du programme sur les entrées de test.

l'entrée `input` est modifiée pour générer un nouveau fichier de test. Une variable `bound` est associée à chaque entrée pour ne pas analyser plusieurs conditions de chemins identiques bien qu'issues d'entrées différentes.

Un exemple d'exécution de cet algorithme est illustré par la figure 6.

3.2 Implémentation

Recherche des conditions de chemin La première étape de l'algorithme est de chercher les path conditions liées à une entrée donnée. Pour ce faire, un outil pour Valgrind a été développé, affichant les contraintes liées aux instructions de branchement conditionnelles dépendant de l'entrée rencontrée sur le chemin d'exécution (un exemple de sortie de cet outil est présenté dans la figure 7). Il repose sur le marquage des données issues de l'entrée suivie, et de la propagation des contraintes dépendant de ces données. L'implémentation du traçage des données marquées (*taint tracing*) est divisée en trois étapes principales :

1. le marquage initial des données issues de la source suivie,
2. la propagation et l'affichage des contraintes associées aux données marquées,
3. la suppression du marquage.

```
def expand_execution(input):
    child_inputs = []

    # execution symbolique et recuperation des
    # contraintes sur l'entree donnee
    pc = compute_path_constraint(input)
    pc = optimize(pc)

    for j in range(input.bound, len(pc)):
        # inversion de la derniere contrainte
        negate(pc[j])
        # resolution des contraintes
        solution = solve(pc)
        if solution:
            # generation d'un nouveau fichier de test
            new_input = modify_bytes(input, solution)
            new_input.bound = j
            child_inputs.append(new_input)

    return child_inputs
```

Fig. 5. Algorithme de génération de nouvelles entrées.

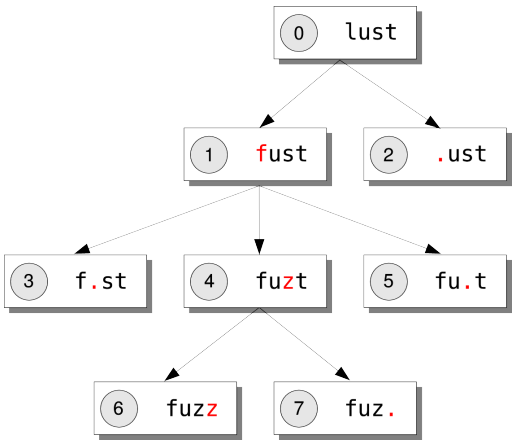

Fig. 6. Graphe de l'exécution de l'algorithme de génération des fichiers de test sur la fonction `strcmp(buffer, "fuzz")`, où `buffer` est une entrée marquée initalisée par la chaîne de caractères `lust`. Les numéros indiquent l'ordre dans lequel les entrées ont été générées. Les caractères rouge sont les caractères différents par rapport à l'entrée précédente et le point représente le caractère nul.

```
% valgrind --tool=fuzzgrind --taint-file=yes --file-filter=/etc/passwd \
  ./target /etc/passwd
...
0x08048e0d: CmpEQ32(And32(8Uto32(GET:I8(PUT(8Uto32(LDle:I8(input(0))))),
            0xff:I32),0x0:I32) => 0
0x08048e16: CmpEQ8(32to8(8Uto32(GET:I8(PUT(8Uto32(LDle:I8(input(0)))))),
            0x2d:I8) => 0
0x08048e26: CmpEQ8(32to8(8Uto32(GET:I8(PUT(8Uto32(LDle:I8(input(0)))))),
            0x2b:I8) => 0
0x08048e39: CmpEQ8(32to8(8Uto32(GET:I8(PUT(8Uto32(LDle:I8(input(0)))))),
            0x72:I8) => 1
...
```

Fig. 7. Extrait du résultat de l'exécution de l'outil sur le programme `target`. Les quatre lignes principales sont constituées de l'adresse de l'instruction de branchement conditionnelle, la contrainte associée, suivi de 1 si le branchement a été pris ou 0 sinon. Dans cet exemple, chaque contrainte compare le premier octet de l'entrée à une constante, et seul le dernier saut a été pris (le premier octet de l'entrée est donc 'r' (0x72)).

Les fichiers et l'entrée standard sont les deux types d'entrée supportées par l'outil. La source d'entrée suivie est spécifiée par la ligne de commande. Si la source d'entrée suivie est l'entrée standard, toutes les données provenant du descripteur de fichier 0 sont marquées. Dans le cas d'un fichier, l'appel système `open` détermine si le descripteur de fichier doit être suivi ou non. Un descripteur de fichier arrête d'être suivi lorsqu'il est fermé par l'appel système `close`. Lorsque l'application cible effectue un appel système à `read` ou `mmap` et que le descripteur de fichier est suivi, les adresses auxquelles les données sont lues ou mmappées sont marquées. À chaque donnée lue marquée est associée une contrainte, correspondant au numéro i de l'octet du fichier lu ou mmapé, et représentée par `input(i)`.

La propagation des contraintes est effectuée par la fonction de l'outil responsable de l'instrumentation de la représentation intermédiaire issue de la phase 3. Les données marquées sont les registres virtuels et temporaires de Valgrind, ainsi que les adresses mémoires et les registres du programme cible. Suivant les instructions de la représentation intermédiaire, une des actions suivantes doit être effectuée :

- Si l'instruction ne dépend pas d'une donnée marquée, celle-ci est ignorée.
- Si l'instruction effectue une opération sur une donnée marquée, le temporaire sauvegardant le résultat doit être marqué, et la contrainte associée à l'opération lui être associée.

– Si l'instruction effectue une opération sur une donnée non marquée, mais le temporaire sauvegardant le résultat est marqué, alors il doit être démarqué.
– Enfin, si une condition dépend d'une donnée marquée, la contrainte associée est affichée.

Ces actions sont effectuées par des fonctions appelées sur chaque instruction de la représentation intermédiaire de Valgrind.

Génération, optimisation et résolution des contraintes Une fois les conditions de chemin affichées, elles sont traduites dans le langage de STP avant d'être optimisées, inversées, puis résolues, comme le montre la figure 8.

```
% cat valgrind_output.txt
CmpEQ8(32to8(8Uto32(LDle:I8(input(0)))),0x72:I8) => 0
CmpEQ8(32to8(8Uto32(LDle:I8(input(1)))),0x00:I8) => 1

% ./convert.py valgrind_output.txt
x0 : BITVECTOR(8);
x1 : BITVECTOR(8);
QUERY(NOT(NOT(x0 = 0h72) AND (x1 = 0h00)));
```

Fig. 8. La sortie de Valgrind est composée de deux conditions associées à des instructions de branchement : le premier octet de l'entrée doit être égal à 0x72, est le second à 0x00. Le premier branchement n'a pas été pris, le second si. Le script `convert.py` convertit ces conditions dans le langage STP.

Toutes ces étapes sont effectuées par un script Python qui parse la sortie de Valgrind afin de récupérer chaque contrainte. Quelques optimisations basiques sont effectuées pour réduire la taille des contraintes et supprimer les doublons, avant d'être converties dans le langage de STP. Les conditions des instructions conditionnelles sont ensuite inversées puis résolues une à une. La résolution des contraintes est effectuée en enregistrant les contraintes dans un fichier, passé en entrée à STP. STP est capable d'afficher un contre-exemple (constitué de valeurs assignées aux variables de la contrainte), lorsque la requête effectuée est invalide. Cette propriété est utilisée pour assigner ces nouvelles valeurs à certains octets de l'entrée. Les contraintes résolues génèrent ainsi de nouveaux fichiers de test, qui découvriront de nouveaux chemins d'exécution.

Détection de fautes L'exécution du programme cible sur les fichiers générés découvre de nouveaux chemins d'exécution, susceptibles de déclencher des bugs pouvant éventuellement être des vulnérabilités.

La détection de fautes est faite par un simple programme utilisant `ptrace` pour examiner tous les signaux reçus par le programme. Si l'un d'entre eux fait partie d'une liste donnée (`SIGSEGV`, `SIGILL`, `SIGABORT`, etc), alors l'entrée de ce programme est considérée comme déclenchant un bug. Il faudra alors vérifier manuellement si c'en est effectivement un, et le cas échéant, analyser si ce bug est exploitable ou non.

Score Le programme associant un score à chaque entrée calcule seulement le nombre de *basic block* exécutés. Ce fonctionnement est temporaire et doit être encore amélioré.

4 Résultats

4.1 Résultats

Actuellement, les fichiers de test générés par Fuzzgrind couvrent entièrement l'espace des entrées de programmes simples, appelant des fonctions de la `libc` (`strcmp`, `atoi`, etc), et effectuant des opérations sur des chaînes de caractères et des entiers.

Des vulnérabilités jusqu'alors inconnues ont été découvertes dans les dernières versions de `readelf` [9] et `swfextract` [10] en moins d'une heure. Ces vulnérabilités ont été découvertes de façon automatique en donnant à Fuzzgrind le programme cible et un fichier valide, et en attendant que les fichiers de test générés déclenchent une erreur.

La version 3.8.2 de la bibliothèque `libtiff` [11] a été fuzzée. Le programme cible (utilisant cette bibliothèque) est `tiffinfo`, et l'entrée initiale est une image tiff de 2x2 pixels, de 961 octets. Une première vulnérabilité est trouvée en 6 minutes. Au cours d'un audit [12] de cette bibliothèque, Tavis Ormandy avait déjà découvert cette vulnérabilité. Délicate à trouver manuellement, elle est à l'origine des exploits [13] [14] permettant le déblocage de la console PSP et de l'Iphone.

[9] http://www.gnu.org/software/binutils/
[10] http://www.swftools.org/swfextract.html
[11] http://www.libtiff.org
[12] http://www.scary.beasts.org/security/tavis_libtiff.txt
[13] http://www.toc2rta.com/?q=node/23
[14] http://www.maxconsole.net/?mode=news&newsid=9516

Chose amusante, les dix crackmes de IOLI [15] sont résolus en quelques secondes, après avoir spécifié des variables d'environnement correctes.

Un nombre plus important de logiciels, à jour ou non, devra être testé afin d'évaluer les capacités de Fuzzgrind à effectivement trouver des vulnérabilités dans des programmes de taille conséquente.

4.2 Performances

Les performances globales de Fuzzgrind dépendent des performances des outils utilisés à chaque étape d'un cycle d'une session, composé de :

1. l'exécution de Valgrind sur une entrée donnée,
2. l'analyse des contraintes par un ensemble de scripts Python,
3. la résolution des contraintes par STP,
4. la détection de fautes éventuelles sur les nouvelles entrées générées,
5. l'assignation d'un score à chaque nouvelle entrée.

Les performances de l'outil de Valgrind, développé pour suivre et propager les conditions de chemin, sont acceptables. Les adresses, les registres, et les temporaires sont actuellement stockés dans des tableaux statiques, qui sont parcourus à quasiment chaque instruction. Leur remplacement par des tables de hachage devrait améliorer les performances.

Les performances de STP sont quant à elles impressionnantes. La figure 9 présente quelques statistiques obtenues par l'exécution de STP sur des contraintes générées par Fuzzgrind :

Fig. 9. Temps d'exécution de STP sur des contraintes générées par Fuzzgrind.

nb de contraintes	taille	temps d'exécution
1	95o	0.003s
102	21,4Ko	0.082s
1000	156,6Ko	0.349s
1251	189,4Ko	0.984s

[15] http://radare.nopcode.org/wiki/index.php?n=Examples.Crackme

Le temps d'exécution de l'outil de détection de fautes est sensiblement le même que celui du programme testé sans ptrace.

L'exécution de l'outil assignant un score à chaque fichier de test est relativement lente, mais cet outil n'est cependant pas définitif.

La figure 10 présente quelques statistiques sur les programmes suivants :
– strcmp : un simple programme faisant appel à la fonction `strcmp`,
– readelf -h input.elf, où input.elf est un binaire ELF strippé de 3Ko,
– tiffinfo input.tiff, où input.tiff est une image TIFF de 961 octets.

Fig. 10. Temps d'exécution de chaque étape du premier cycle d'une session, obtenues avec un processeur Intel Core 2 Duo à 2,40Ghz et possédant 3Go de RAM.

Étape	1	2	3	4	5
`strcmp`	0s	0s	0s	0s	0s
`readelf`	3s	36m	1h10	1s	56s
`tiffinfo`	1s	5s	7m	1s	7s

4.3 Limitations

L'analyse de programmes conséquents entraîne une explosion du nombre de chemins d'exécution possible. Il est possible de limiter le nombre de contraintes à analyser, mais la couverture des chemins sera forcément réduite. Si le programme à analyser contient des fonctions cryptographiques (`hash md5` par exemple), il est envisageable de les désactiver (à moins qu'elles ne soient la cible des tests).

Par ailleurs, Fuzzgrind ne suit que les contraintes sur des nombres entiers (de taille inférieure ou égale à 32 bits). Les contraintes sur les nombres flottants ne sont pas suivies.

4.4 Améliorations envisageables

Valgrind présente le très grand avantage d'être un logiciel libre, en plus d'avoir des performances correctes. Bien que supportant une multitude d'architectures, il ne fonctionne cependant que sur des systèmes d'exploitation compatibles UNIX, limitant les

applications pouvant être testées par Fuzzgrind. Intel a de son coté développé un outil d'instrumentation binaire dynamique, PIN [16], supportant la plupart des architectures Intel et les systèmes d'exploitation Linux et Windows. D'après les *benchmarks* publiés par Intel, ses performances seraient bien meilleures que celles de Valgrind. La licence de son noyau est cependant propriétaire (mais ses outils d'instrumentation sont libres).

Fuzzgrind suit actuellement les entrées de type fichier et entrée standard. Le suivi de ces entrées est simple puisque les fichiers ouverts peuvent être filtrés par leur nom, et l'entrée standard est toujours associée au descripteur de fichier 0. Il serait intéressant d'étendre les entrées supportées aux entrées réseaux, ce qui est légèrement plus compliqué. Cette amélioration est envisageable par le suivi des appels système `accept`, `connect`, et `socketpair` ; et le filtrage des descripteurs de fichiers résultant de ces appels grâce à l'appel système `getsockbyname` (retournant des informations sur les couple adresse IP/port source et destination).

L'outil de calcul de score des nouvelles entrées repose actuellement sur le plugin Lackey [17] de Valgrind, destiné à faire des statistiques sur l'exécution d'un programme. Sa fonctionnalité de calcul *approximatif* du nombre de blocs basiques exécutés est utilisé pour calculer les scores. Les performances et la précision de ce plugin d'exemple restent désirables, et la fonction de calcul de score doit donc être améliorée. Bien que conséquent à implémenter, un outil déterminant correctement la couverture de code de l'exécution de chaque nouvelle entrée serait assurément intéressante.

Enfin, les performances globales de l'outil peuvent encore être améliorées. Un moteur de simplification des contraintes ainsi qu'un système de cache augmenteraient la vitesse d'exécution.

5 Conclusion

Bien que Fuzzgrind soit encore en développement intensif, les quelques vulnérabilités découvertes sont encourageantes et confirment que ses concepts sont prometteurs. Son implémentation montre que la réalisation d'outils de fuzzing complètement automatiques est possible, offrant la possibilité de tester la sécurité de logiciels sans aucun pré-requis.

[16] http://www.pintool.org
[17] http://www.valgrind.org/docs/manual/lk-manual.html

Références

1. Vuagnoux, M. : Autodafé : an act of software torture. http://autodafe.sourceforge.net/paper/autodafe.pdf (2006)
2. Molnar, D., Wagner, D. : Catchconv : Symbolic execution and run-time type inference for integer conversion errors. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-23.pdf
3. Zalewski, M. : Bunny the fuzzer. http://code.google.com/p/bunny-the-fuzzer/
4. Drewry, W., Ormandy, T. : Flayer : exposing application internals. In : WOOT '07 : Proceedings of the first USENIX workshop on Offensive Technologies. (2007) 1–9
5. Godefroid, P., Klarlund, N., Sen, K. : Dart : directed automated random testing. http://cm.bell-labs.com/who/god/public_psfiles/pldi2005.pdf (2005)
6. Sen, K., Marinov, D., Agha, G. : Cute : a concolic unit testing engine for c. http://srl.cs.berkeley.edu/~ksen/pubs/paper/C159-sen.pdf (2005)
7. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R. : Exe : automatically generating inputs of death. http://www.stanford.edu/~engler/exe-ccs-06.pdf (2006)
8. Godefroid, P., Levin, M., Molnar, D. : Automated whitebox fuzz testing. http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf (2008)
9. Nethercote, N., Seward, J. : Valgrind : a framework for heavyweight dynamic binary instrumentation. (2007)
10. Ganesh, V., Dill, D.L. : A Decision Procedure for Bit-Vectors and Arrays. PhD thesis (2007)
11. Brumley, D., Poosankam, P., Song, D., Zheng, J. : Automatic patch-based exploit generation is possible : Techniques and implications. http://www.cs.cmu.edu/~dbrumley/pubs/apeg.pdf (April 2008)