# INF889A

# Analyse de programmes pour la sécurité logicielle

Chapitre 1 - Fuzzing

---

Quentin Stiévenart

Hiver 2024

# Référence

The Fuzzing Book (2023)

# Origine du fuzzing

Prof. Barton Miller (University of Wisconsin), en 1988:

- connecté via ligne téléphone 1200 baud à l'ordinateur de l'université, une nuit orageuse
- entrées des commandes Unix corrompues
- crash observés

Projet proposé aux personnes étudiantes :

> *The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. [...] First, you will build a fuzz generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them.*

# Origine du fuzzing

## Miller et al., CACM 1990

Résultats :

> *About a third of the UNIX utilities they fuzzed had issues – they crashed, hung, or otherwise failed when confronted with fuzzing input*

Miller, Fredriksen, & So. (1990). An Empirical Study of the Reliability of UNIX Utilities. CACM.

## Avgerinos et al., CACM 2014

Avgerinos et al. (2014). Automatic Exploit Generation. CACM.

```
input=$(openssl rand -hex 6676)
for program in /usr/bin/*; do
  for letter in {a..z} {A..Z}; do
    timeout -s 9 1s ${program} -${letter} ${input}
  done
done
```

## Avgerinos et al., CACM 2014

Sur l'installation de base de Debian 6.0.5 (1168 exécutables) :

- 13 minutes
- 756 crashes
- 52 bugs identifiés
- 29 programmes affectés
- 5 vulnérabilités qui permettent un contournement de flot de contrôle
- 4 exploits générés automatiquement par l'outil proposé dans l'article (*Mayhem*)

# Fuzzing : dimensions

- Comment générer des entrées ?
  - Fuzzers aléatoires
  - Fuzzers basés sur la mutation
  - Fuzzers basés sur les grammaires
- Quel niveau de connaissance du format d'entrée ?
  - Fuzzer non-structurés
  - Fuzzer structurés
- Quel niveau de connaissance du programme testé ?
  - Black-box fuzzing
  - Grey-box fuzzing
  - White-box fuzzing
- Comment détecter un état indésirable ?
- Comment minimiser une entrée qui génère un état indésirable ?
- Comment savoir quand s'arrêter ?

# Fuzzing naïf : dimensions

```
input=$(openssl rand -hex 6676)
for program in /usr/bin/*; do
  for letter in {a..z} {A..Z}; do
    timeout -s 9 1s ${program} -${letter} ${input}
  done
done
```

- Fuzzer basé sur la génération purement aléatoire
- Fuzzer non-structuré : pas d'hypothèse sur l'entrée
- Fuzzer black-box : pas de connaissance du programme
- État indésirable = crash
- Aucune minimisation d'entrée
- Condition d'arrêt basique

# Contexte : automation de tests

```
test(program, budget):
    faults = []
    while budget.is_sufficient():
        input = generate_input(program)
        report = run_and_monitor(program, input)
        if report.contains_failure():
            fault.add(locate_fault(program, report))
    while budget.is_sufficient():
        fault = pop_most_severe(fault)
        if generate_exploit:
            exploit = exploit(program, fault)
        if fix_program:
          program = fix(program, fault)
```

Marcozzi M. Finding Software Vulnerabilities with Fuzzing, STT&V 2023

# Contexte : automation de tests

Plusieurs problèmes indépendants :

- `generate_input`: problème de génération d'entrées
- `run_and_monitor` : problème de l'oracle de test
- `locate_fault` : problème de localisation d'erreur
- `pop_most_severe` : problème de priorisation de bug
- `exploit` : problème de génération d'exploit
- `fix` : problème de réparation automatique de programme

Le fuzzing concerne la **génération d'entrée**, et nécessite un **oracle de test**

Les autres éléments sont des problèmes de recherche connexes

# Fuzzing : détecter des états indésirables

# Détection d'état indésirable : plantage du programme

## Regarder le code d'erreur

```
timeout -s 9 1s ${program} -${letter} ${input} \
  || echo 'Crash!'
```

Facile à mettre en place ! Mais :

- peut-être que le programme a été corrompu sans planter
  $\rightarrow$ faux négatif
- peut-être que le programme a besoin de plus de temps
  $\rightarrow$ faux positif

# Détection d'état indésirable : échec d'assertion

**Activer les assertions**

On souhaite détecter un état indésirable au plus tôt

Dans Firefox, « *fuzzing JS and DOM has found about 4000 assertion bugs, including about 300 security bugs* »

# Importance d'activer les assertions

code/assert.c

```c
int main(int argc, char** argv) {
    char *buf = malloc(100 * sizeof(char));
    memset(buf, 42, 100);

    int index = atoi(argv[1]);
    char val = buf[index];
#if DEBUG
    assert(val == 42);
#endif

    free(buf);
    return val;
}
```

# Importance d'activer les assertions

```
$ clang assert.c -o assert
$ ./assert 10 ; echo $?
42
$ ./assert 100 ; echo $?
0
$ clang assert.c -o assert -DDEBUG
$ ./assert 10 ; echo $?
42
$ ./assert 100 ; echo $?
assert: assert.c:12: main: Assertion `val == 42' failed.
Aborted (core dumped)
```

# Utiliser des *checkers*

Options des compilateurs, par exemple dans LLVM :

- `-fsanitize=address`: “AddressSanitizer, a memory error detector”
- `-fsanitize=thread`: “ThreadSanitizer, a data race detector”
- `-fsanitize=memory`: “MemorySanitizer, a detector of uninitialized reads”
- ...

# Utilisation d'ASan

code/assert.c

ASan vérifie les accès mémoire. Plus de détails au chapitre 2.

```
$ clang assert.c -o assert -fsanitize=address
$ ./assert 10 ; echo $?
42
$ ./assert 100 ; echo $?
=================================================================
==413879==ERROR: AddressSanitizer: heap-buffer-overflow on
READ of size 1 at 0x60b0000000a4 thread T0
    #0 0x561f884e9253 in main (/sync/INF889A/slides/code/a
    #1 0x7f0e47045ccf  (/usr/lib/libc.so.6+0x27ccf) (Build
    #2 0x7f0e47045d89 in __libc_start_main (/usr/lib/libc.
    #3 0x561f884e90d4 in _start (/sync/INF889A/slides/code/

...
```

# Tests différentiels

Nécessite une implémentation de référence, ou plusieurs implémentations à comparer.

Chen, & Su. (2015). Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. ESEC/FSE.

> *In our work, we employ the generated mucerts to test commonly used SSL/TLS implementations (including OpenSSL [1], PolarSSL [3], Gnutls [4], NSS [2], CyaSSL [5], and MatrixSSL [6]) and web browsers (including Google's Chrome [7], Mozilla's Firefox [8], and Microsoft's Internet Explorer [9]) and then compare the validation results. Any behavior discrepancies among these implementations become oracles for finding flaws in their certificate validation code.*

# Détection de fuite d'information

code/secret.c

```
void secret() {
    char *secret = malloc(100 * sizeof(char));
    for (int i = 0; i < 100; i += 6) {
        strncpy(secret+i, "secret", 6);
    }
    free(secret);
}
```

# Détection de fuite d'information

code/secret.c

```c
int main(int argc, char** argv) {
    secret();

    char *buf = malloc(90 * sizeof(char));
    memset(buf, 'A', 90);
    memcpy(buf, argv[1], strlen(argv[1]));

    int length = atoi(argv[2]);
    printf("%.*s\n", length, buf);

    free(buf);
    return 0;
}
```

# Détection de fuite d'information

```
$ ./secret hat 3
hat
$ ./secret hat 100
hatAAAAAAAAAAAAAAAAAAAA...AAAAAAAAAAsecret
```

# Génération d'entrées

# Génération aléatoire

code/fuzz.cpp

```
std::string fuzzer(int max_length = 100,
                   int char_start = 32,
                   int char_range = 96) {
    int string_length = rand() % (max_length + 1);
    std::string out = "";
    for (int i = 0; i < string_length; ++i) {
        char random_char = char_start + rand() % char_range;
        out += random_char;
    }
    return out;
}
```

# Génération aléatoire

Scénario : on souhaite fuzzer un parseur d'URL

Chances de générer `http://` ou `https://` : $\frac{1}{range^7} + \frac{1}{range^8}$

Avec $range = 96$, cela fait $1.34 \cdot 10^{-14}$

Il faudra en moyenne $74\ 370\ 059\ 689\ 055$ itérations pour générer un préfixe valide.

```
$ clang++ -O3 fuzz.cpp -o fuzz
$ ./fuzz
1 iteration took 1.00785e-06 seconds
I need 7.56915e+07 seconds
```

$=$ 2.4 années

# Génération basée sur les mutations

On peut partir d'un ensemble d'entrées correctes

Ensuite on applique des transformations, par exemple

- ajouter des caractères
- supprimer des caractères
- modifier des caractères

# Génération basée sur les mutations

code/mutate.cpp

```cpp
std::mt19937 rg;

std::string mutate(const std::string& s) {
    std::vector<std::string (*)(const std::string&)> mutators = {
        insertRandomCharacter,
        deleteRandomCharacter,
        flipRandomCharacter
    };

    std::uniform_int_distribution<int> dist(0, mutators.size() - 1);
    int index = dist(rg);
    return mutators[index](s);
}
```

# Génération basée sur les mutations

code/mutate.cpp

```
std::string insertRandomCharacter(const std::string& s) {
    std::uniform_int_distribution<int> distPos(0, s.length());
    std::uniform_int_distribution<int> distChar(32, 127);
    int pos = distPos(rg);
    char randomChar = distChar(rg);
    return s.substr(0, pos) + randomChar + s.substr(pos);
}
```

# Génération basée sur les mutations

code/mutate.cpp

```
std::string deleteRandomCharacter(const std::string& s) {
    if (s.empty()) return s;

    std::uniform_int_distribution<int> dist(0, s.length() - 1);
    int pos = dist(rg);
    return s.substr(0, pos) + s.substr(pos + 1);
}
```

# Génération basée sur les mutations

code/mutate.cpp

```
std::string flipRandomCharacter(const std::string& s) {
    if (s.empty()) return s;

    std::uniform_int_distribution<int> distPos(0, s.length() - 1

    int pos = distPos(rg);
    char c = s[pos];
    int bit = 1 << (rg() % 7);
    char new_c = static_cast<char>(c ^ bit);

    return s.substr(0, pos) + new_c + s.substr(pos + 1);
}
```

# Génération basée sur les mutations

```
$ ./mutate
1 iteration took 6.00312e-06 seconds
Input validity: 0.6174
```

62% des entrées sont valides

# Génération basée sur les grammaires

Si on a une entrée structurée (par exemple un langage), on peut se baser sur la grammaire pour générer notre entrée.

```
<url> ::= <scheme>://<authority><path><query>
<scheme> ::= http | https | ftp | ftps
<authority> ::= <host> | <host>:<port> | <userinfo>@<host> | <userinfo>
<host> ::= hello.world
<port> ::= 80 | 8080 | <nat>
<nat> ::= <digit> | <digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<userinfo> ::= user:password
<path> ::= eps | / | /<id>
<id>  ::= abc | def | x<digit><digit>
<query> ::= eps | ?<params>
<params> ::= <param> | <param>&<params>
<param> ::= eps | <id>=<id> | <id>=<nat>
eps ::= chaîne vide
```

# Génération basée sur les grammaires

code/grammar.cpp

```
std::string query() {
    return choice({"", "?" + params(randInt(1, 4))});
}

std::string path() {
    return choice({"", "/", "/" + id()});
}

std::string port() {
    return choice({"80", "8080", nat(1, 65535) });
}

std::string scheme() {
    return choice({"http", "https", "ftp", "ftps"});
}

std::string url() {
    return scheme() + "://" + authority() + path() + query();
}
```

# Génération basée sur les grammaires

```
$ ./grammar | grep -cvE "^(ht|f)tps?://"
0
```

100% des entrées sont valides !

# Évaluation d'un fuzzer

# Fuzzing : mesurer l'efficacité

Comment mesurer l'efficacité d'un fuzzer ?

- nombre de bugs trouvés : cela dépend de la présence de bugs
- comparaison avec d'autres fuzzers : trouve-t-on un bug non détecté par les autres ?
- couverture de code : quelle portion du programme est testée ?

# Calcul de la couverture

```
$ clang -fprofile-instr-generate -fcoverage-mapping \
    yuarel.c yuarel_client.c -o yuarel
$ ./yuarel http://uqam.ca
$ llvm-profdata merge -sparse default.profraw \
    -o default.profdata
$ llvm-cov show ./yuarel -instr-profile default.profdata
```

Documentation LLVM : Source-Based Code Coverage

# Script d'évaluation

code/evaluateFuzzer.bash

```
#!/usr/bin/env bash
function evaluateFuzzer() {
  name="$1"
  mkdir -p "$name-profraw"
  "$name" | while read -r input; do
    LLVM_PROFILE_FILE="$(mktemp "$name-profraw/XXXXXX.profraw")" \
        ./yuarel "$input"
  done
  llvm-profdata merge -sparse "$name"-profraw/*.profraw -o "$name".prof
  llvm-cov report ./yuarel -instr-profile "$name".profdata
}
```

# Couverture d'un black-box fuzzer aléatoire

```
$ evaluateFuzzer ./fuzz
```

35.14% de couverture de branches sur `yuarel.c`

# Couverture d'un black-box fuzzer basé sur les mutation

```
$ evaluateFuzzer ./mutate
```

56.93% de couverture de branches sur `yuarel.c`

# Couverture d'un black-box fuzzer basé sur les grammaires

```
$ evaluateFuzzer ./grammar
```

40.54% de couverture de branches sur `yuarel.c`

# Le tout ensemble

Si on combine maintenant les trois

- fuzzing aléatoire
- fuzzing à partir de mutations
- fuzzing à partir de grammaires

```
$ function allFuzzers() { ./fuzz; ./mutate; ./grammar; }
$ evaluateFuzzer allFuzzers
```

59.46% de couverture de branches sur `yuarel.c`

# Greybox fuzzing

# Greybox fuzzing

On a traité notre programme comme une boite noire :

- aucune hypothèse sur l'implémentation
- génération entièrement aléatoire

On va maintenant "ouvrir" cette boîte pour guider le fuzzer

# Coverage-guided fuzzing

Idée : on mute les entrées qui mène à de la nouvelle couverture

Structure :

- on a une population de base
- on mute un élément de notre population
- si cela amène à de la nouvelle couverture, on l'ajoute à la population

# Coverage-guided fuzzing

code/coverage.cpp

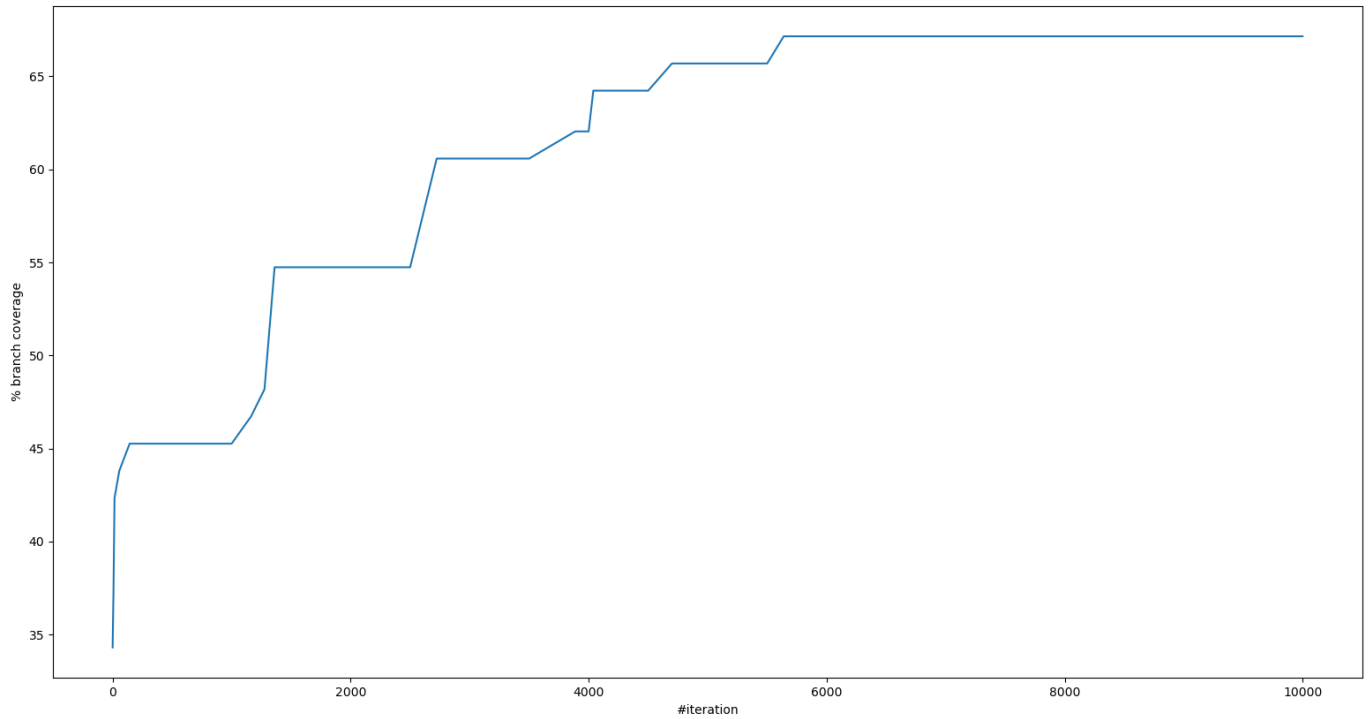
```
    setenv("LLVM_PROFILE_FILE", llvmProfileFile.c_str(), 1);
    exec1("./yuarel", candidate);
}

double getCoverage() {
    execShell("llvm-profdata merge -sparse data/*.profraw -o data.profd
    std::string coverage = execShell("llvm-cov report ./yuarel -instr-p
    return std::stod(coverage);
}

int main() {
    rg.seed(std::random_device()());
    std::vector<std::string> population = {"http://hello.world", "http:
    double previousCoverage = 0.0;
    for (int iteration = 0; iteration < 10000; iteration++) {
        std::string candidate = mutate(randomElement(population));
        executeWithCandidate(iteration, candidate);
        double currentCoverage = getCoverage();
```

# Coverage-guided fuzzing

Évolution de la couverture :



# Coverage-guided fuzzing en pratique

C'est l'idée derrière fonctionne **AFL**

- Instrumentation pour mesurer la couverture :
  `AFL_USE_ASAN=1 CC=afl-clang-fast make test`
- Coverage-guided greybox fuzzing :
  `afl-fuzz -i input/ -o results --`
  `./test_external_rom @@ 100`

Autres éléments mis en place par AFL (cf. doc) :

- *power schedule* : priorise les entrées courtes, rapides, et qui augmentent souvent la couverture
- *fork server* : pour n'`exec` le process qu'une seule fois
- *instrumentation binaire* avec QEMU
- *afl-tmin* pour minimiser les entrées donnant des erreurs
- ...

# Directed fuzzing

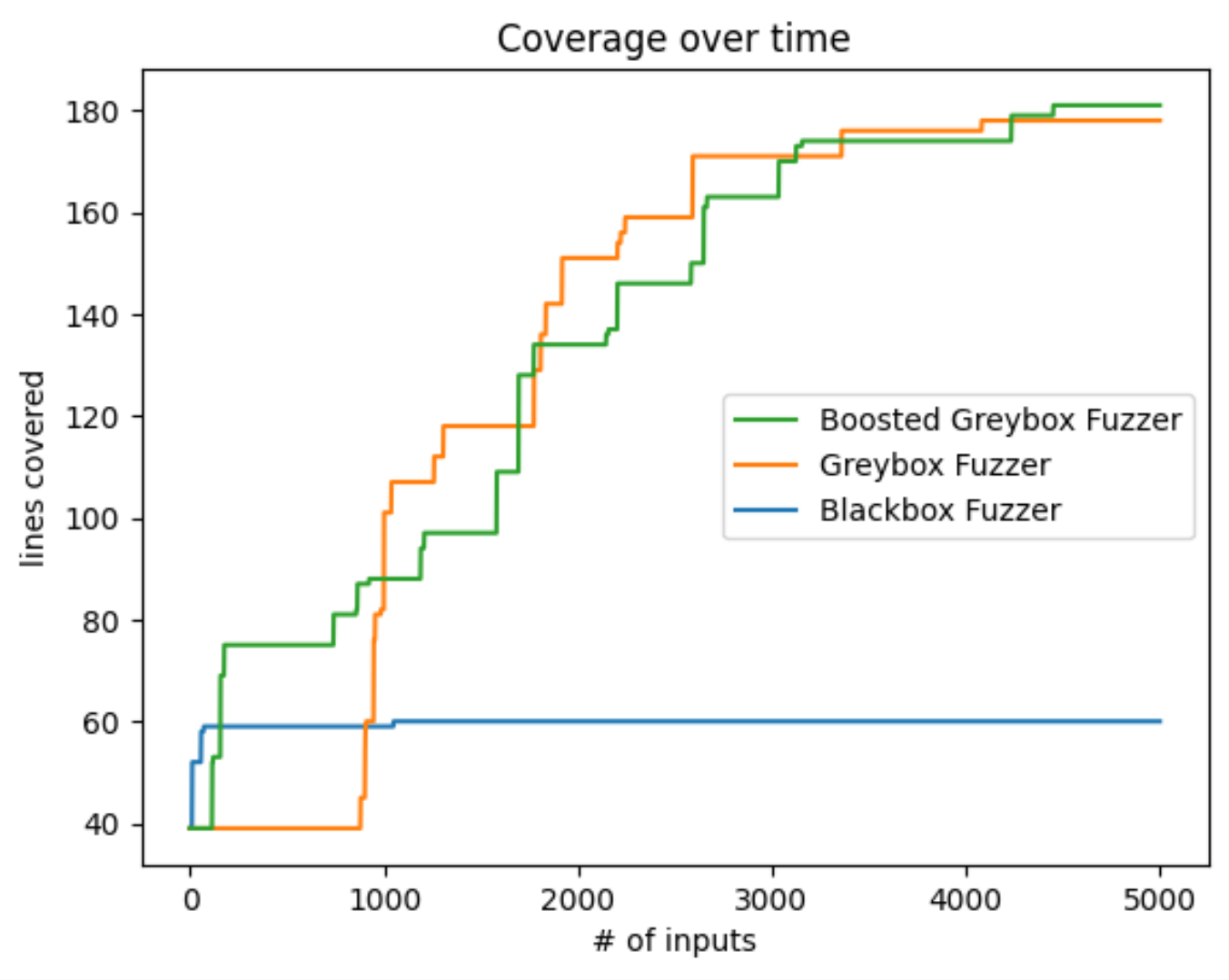
Autre approche de *greybox* fuzzing :

- on souhaite atteindre un certain point dans le programme
- on minimise la distance entre le chemin exploré et le point voulu
- *grey* = l'analyse sur le programme est minime

Exemple du *Fuzzing Book* :

- résoudre un labyrinthe
- priorisation des entrées menant proche de la fonction cible dans le graphe d'appel

# Blackbox vs. greybox fuzzing




Voir Fuzzing Book : Greybox Fuzzer pour plus d'informations.

# Autres considérations

# White-box fuzzing

On peut utiliser des techniques d'analyse avancées pour capturer l'information.

- demande plus de ressources
- efficace pour trouver des bugs situés en profondeur

Typiquement fait avec le l'**analyse symbolique** (cf. Chap. 3)

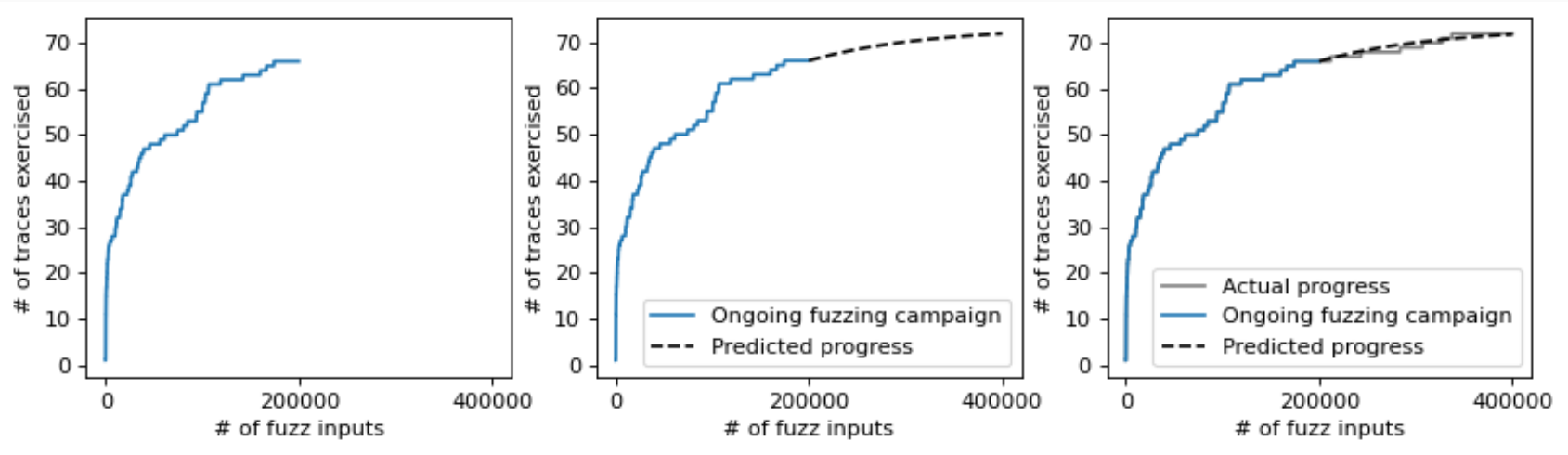
Godefroid. (2020). Fuzzing: Hack, Art, and Science. CACM.

> *the top finalists of the DARPA Cyber Grand Challenge, a competition for automatic security vulnerability detection, exploitation and repair, all included some form of whitebox fuzzing with symbolic execution and constraint solving in their solution*

# Quand s'arrêter

Mesures simples :

- avoir un budget (de temps par exemple)
- viser un certain niveau de couverture
- techniques d'estimations de l'amélioration de la couverture




Fuzzing Book : When To Stop Fuzzing?

# Minimisation d'entrée

Une fois une erreur trouvée, on souhaite minimiser l'entrée pour avoir un test minimal

## Delta-debugging

- Adapté aux entrées peu structurées
- Par exemple par recherche dichomotique
  - On sépare l'entrée en deux (dépend du format d'entrée)
  - On ne garde que la moitié qui mène à l'erreur, et on recommence
  - On s'arrête quand aucune moitié ne mène à l'erreur

## Simplification de grammaire

- Adapté aux entrées structurées
- On simplifie une sous-structure de l'entrée
- Exemple : réduction de sous-arbres dans un AST

# Garanties

- Un bug trouvé est un bug réel $\rightarrow$ **pas de faux positifs**
  - Mais parfois, ce bug n'est pas atteignable en pratique
- Pas de garantie de trouver un bug existant $\rightarrow$ **présence de faux négatifs**
- On fait une **sous-approximation** pour répondre à « *Ce programme contient-il des erreurs* ? »

# Limitations

- Le générateur est crucial
- Besoin de beaucoup de temps
- Tendance à trouver des erreurs “simples”

  $\rightarrow$ le whitebox fuzzing peut aller plus loin

```
if (x == 0x12345678) {
  assert(false);
}
```

# Retour sur les dimensions

- Comment générer des entrées ?
  - Fuzzers aléatoires
  - Fuzzers basés sur la mutation
  - Fuzzers basés sur les grammaires
- Quel niveau de connaissance du format d'entrée ?
  - Fuzzer non-structurés
  - Fuzzer structurés
- Quel niveau de connaissance du programme testé ?
  - Black-box fuzzing
  - Grey-box fuzzing
  - White-box fuzzing
- Comment détecter un état indésirable ?
- Comment minimiser une entrée qui génère un état indésirable ?
- Comment savoir quand s'arrêter ?

# Fuzzing : articles (I)

Détection de vulnérabilités :

- Li et al. (2019). Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. ESEC/FSE.
- Park et al. (2022). FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. Usenix Security.
- Yuan et al. (2023). DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. Usenix Security.
- Kim et al. (2022). FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing. Usenix Security.
- Aafer et al. (2021). Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing. Usenix Security.

# Fuzzing : articles (II)

Anti-fuzzing :

- Güler et al. (2019). AntiFuzz: Impeding Fuzzing Audits of Binary Executables. Usenix Security.
- Jung et al. (2019). Fuzzification: Anti-Fuzzing Techniques. Usenix Security.
- Zhou et al. (2022). No-Fuzz: Efficient Anti-Fuzzing Techniques. SecureComm.

# Fuzzing : articles (III)

Techniques de fuzzing :

- Zhang et al. (2023). Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing. Usenix Security.
- Li and Su. (2023). Accelerating Fuzzing through Prefix-Guided Execution. OOPSLA.
- Wüstholz and Christakis. (2020). Targeted Greybox Fuzzing with Static Lookahead Analysis. ICSE.
- Srivastava and Payer. (2021). Gramatron: Effective Grammar-Aware Fuzzing. ISSTA.

# Fuzzing : articles (IV)

Applications spécifiques

- Mansur et al. (2020). Detecting Critical Bugs in SMT Solvers using Blackbox Mutational Fuzzing. ESEC/FSE.
- Even-Mendoza et al. (2023). GrayC: Greybox Fuzzing of Compilers and Analysers for C. ISSTA.

# Pratique

# Pratique

Au choix :

- Inspecter la couverture pour comprendre pourquoi on bloque à 65% de couverture. Améliorer le fuzzer pour qu'il aille plus loin.
- Utiliser un des outils listés et analyser un projet au choix.
  - Quel est le point d'entrée ?
  - Y a-t-il des assertions à activer ?
  - Comment l'outil détecte un état d'erreur ?
  - Trouvez-vous des bugs ?

# Outils de fuzzing

- AFL++ : amélioration de AFL, avec de nombreux plugins
- libfuzzer : fuzzer basé sur la couverture, intégré au processus testé
- honggfuzz : fuzzer basé sur la couverture, bas niveau (utilise des compteurs matériels et `ptrace`)
- Radamsa : fuzzer black-box générique
- Grammarinator : fuzzer de grammaires
- WSFuzzer : fuzzer de services web