

afl-cov-fast – SSTIC 2025

Génération de couverture de code pour AFL++



Jean-Romain Garnier – Airbus security lab
04/06/2025

AIRBUS

1

Introduction au fuzzing

2

Couverture de code

3

Présentation de l'outil

4

Conclusion

Contexte

Objectif de la présentation

```
american_fuzzy_lop ++2.65d (libpng_harness) [explore] {0}

process timing
  run time : 0 days, 0 hrs, 0 min, 43 sec
  last new path : 0 days, 0 hrs, 0 min, 1 sec
  last uniq crash : none seen yet
  last uniq hang : none seen yet

cycle progress
  now processing : 261*1 (37.1%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : splice 14
  stage execs : 31/32 (96.88%)
  total execs : 2.55M
  exec speed : 61.2k/sec

fuzzing strategy yields
  bit flips : n/a, n/a, n/a
  byte flips : n/a, n/a, n/a
  arithmetics : n/a, n/a, n/a
  known ints : n/a, n/a, n/a
  dictionary : n/a, n/a, n/a
  havoc/splice : 506/1.05M, 193/1.44M
  py/custom : 0/0, 0/0
  trim : 19.25%/53.2k, n/a

overall results
  cycles done : 15
  total paths : 703
  uniq crashes : 0
  uniq hangs : 0

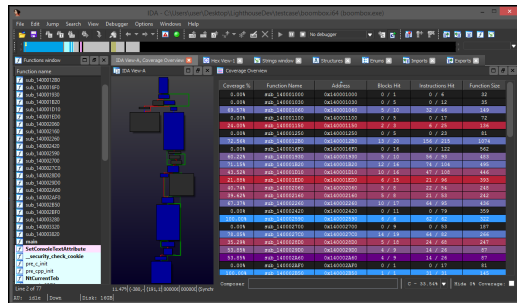
map coverage
  map density : 5.78% / 13.98%
  count coverage : 3.30 bits/tuple

findings in depth
  favored paths : 114 (16.22%)
  new edges on : 167 (23.76%)
  total crashes : 0 (0 unique)
  total tmouts : 0 (0 unique)

path geometry
  levels : 11
  pending : 121
  pend fav : 0
  own flnds : 699
  imported : n/a
  stability : 99.88%

[cpu000: 12%]
```

D'une campagne AFL++...



... À une visualisation de couverture de code

Contexte

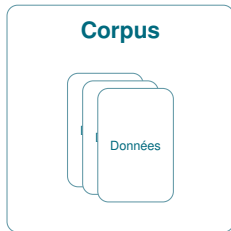
Principe du fuzzing



Fuzzer

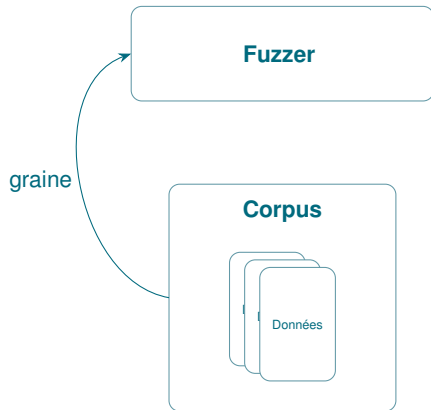
Contexte

Principe du fuzzing



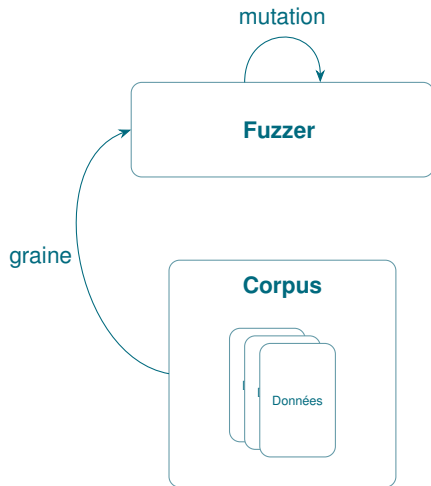
Contexte

Principe du fuzzing



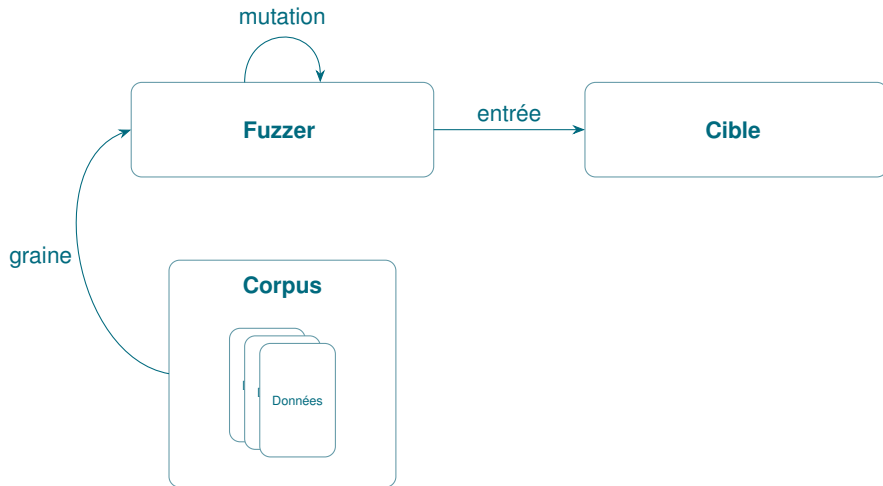
Contexte

Principe du fuzzing



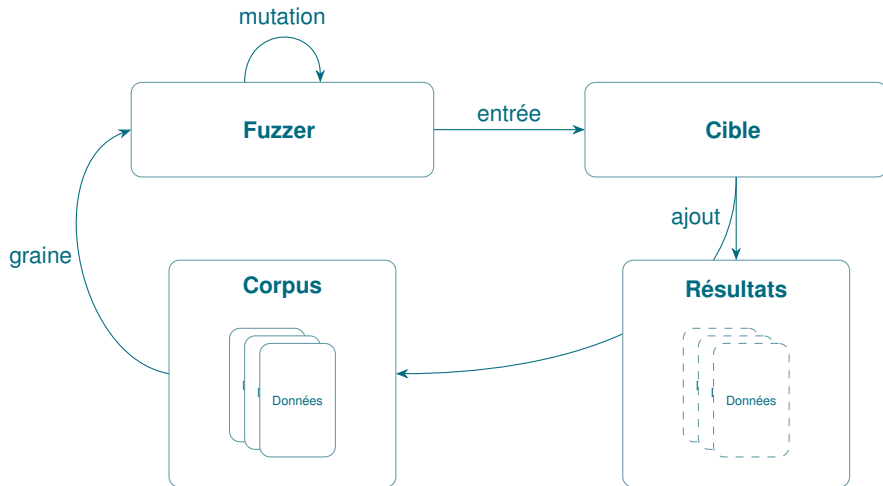
Contexte

Principe du fuzzing



Contexte

Principe du fuzzing



Fuzzer

AFL++



American Fuzzy Loop plus plus (AFL++)

Code : <https://github.com/AFLplusplus/AFLplusplus>

Documentation : <https://aflplus.plus>

Fuzzer

AFL++



American Fuzzy Loop plus plus (AFL++)

Code : <https://github.com/AFLplusplus/AFLplusplus>

Documentation : <https://aflplus.plus>

- Fuzzer guidé par la couverture de code
- Supportant les cibles avec ou sans code source
- Capable de détecter une variété de comportements (via ASAN, MSAN, UBSAN, etc.)

Fuzzer

AFL++



American Fuzzy Loop plus plus (AFL++)

Code : <https://github.com/AFLplusplus/AFLplusplus>

Documentation : <https://aflplus.plus>

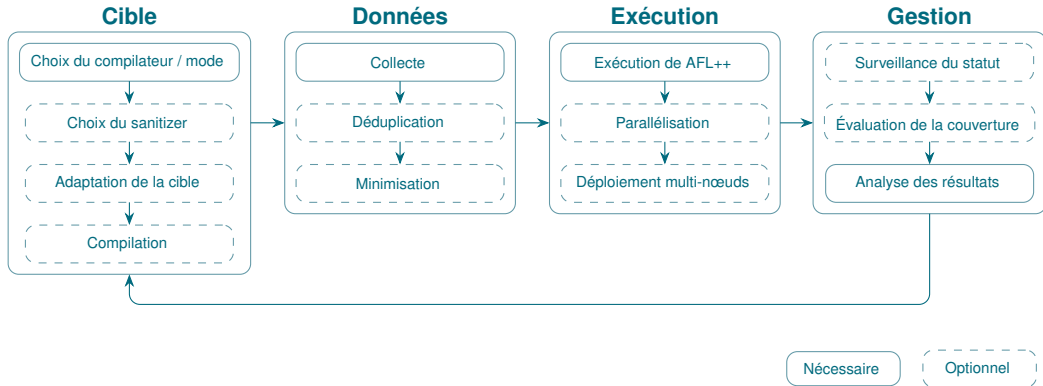
- Fuzzer guidé par la couverture de code
- Supportant les cibles avec ou sans code source
- Capable de détecter une variété de comportements (via ASAN, MSAN, UBSAN, etc.)

By avg. score		By avg. rank	
libafl	98.63	afplusplus	1.76
afplusplus	95.40	libafl	1.81
libfuzzer	85.39	libfuzzer	2.95
afl	82.39	afl	3.38

<https://google.github.io/fuzzbench>

Fuzzer

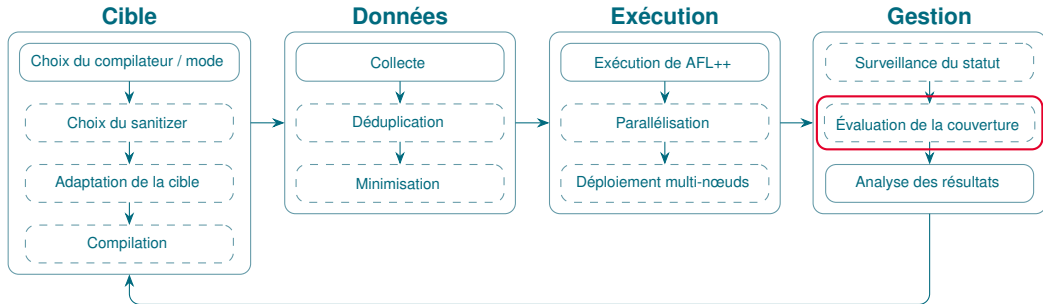
Principe détaillé du fuzzing (AFL++)



<https://aflplus.plus/docs>

Fuzzer

Principe détaillé du fuzzing (AFL++)



Nécessaire

Optionnel

<https://aflplus.plus/docs>

1 Introduction au fuzzing

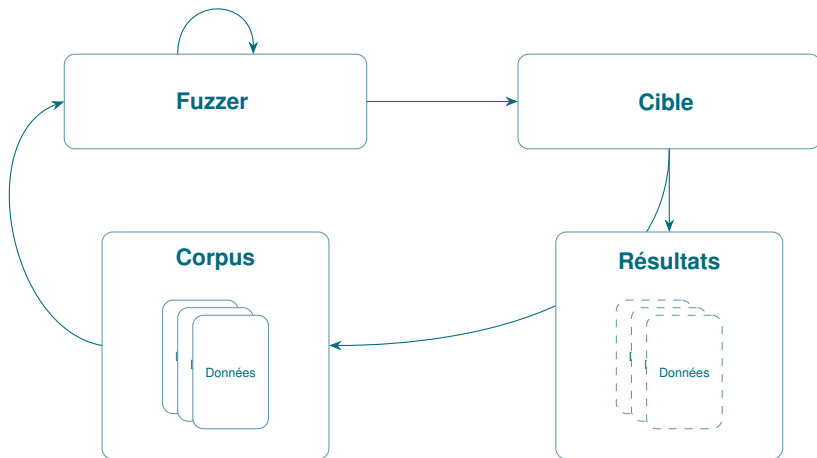
2 Couverture de code

3 Présentation de l'outil

4 Conclusion

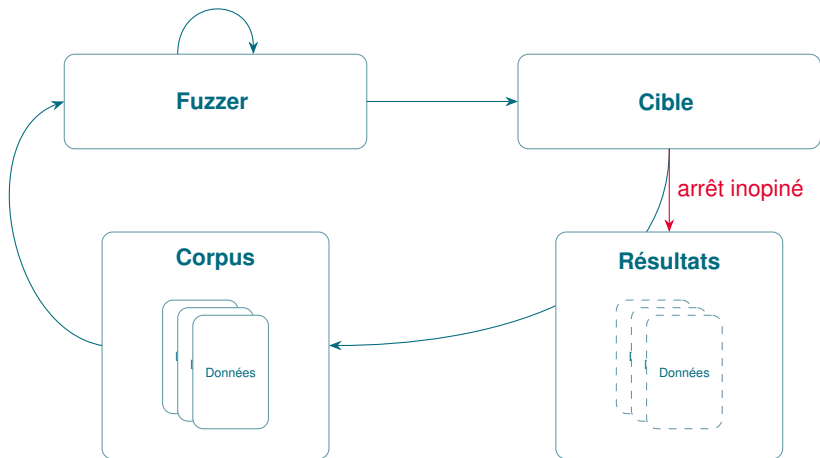
Mesure de la couverture de code

Description du besoin



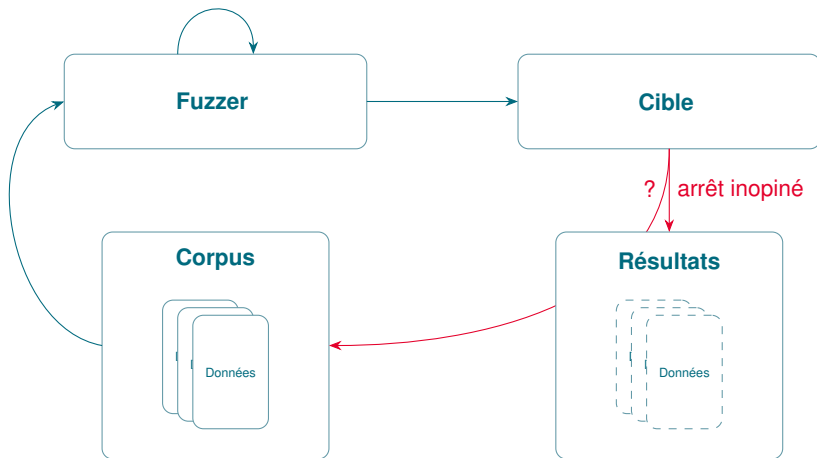
Mesure de la couverture de code

Description du besoin



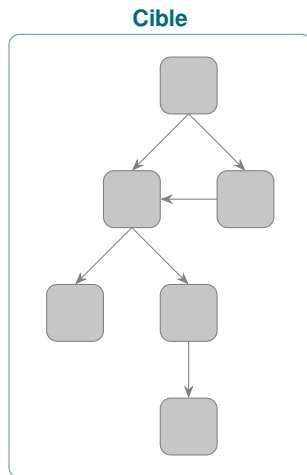
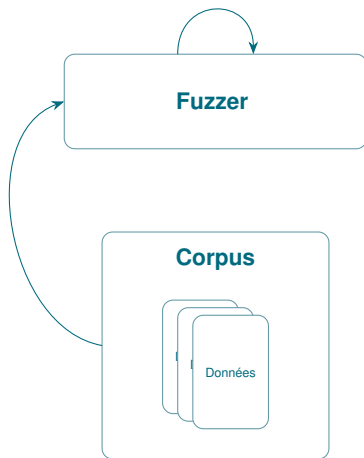
Mesure de la couverture de code

Description du besoin



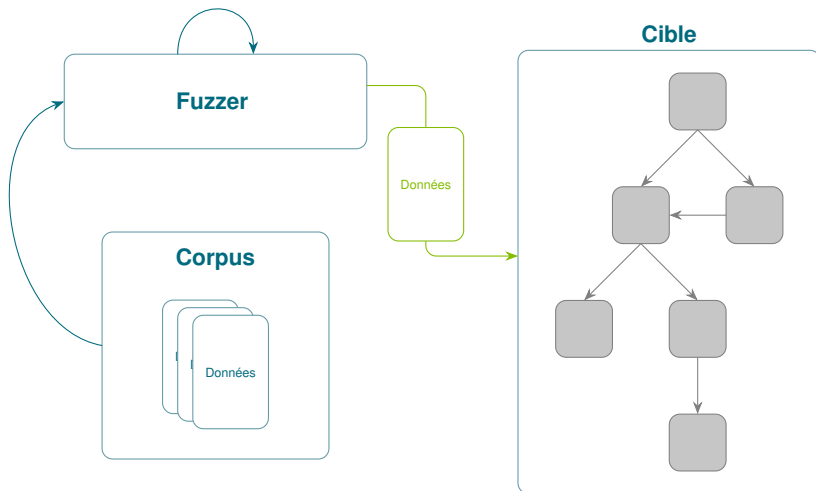
Mesure de la couverture de code

Approche : blocs élémentaires



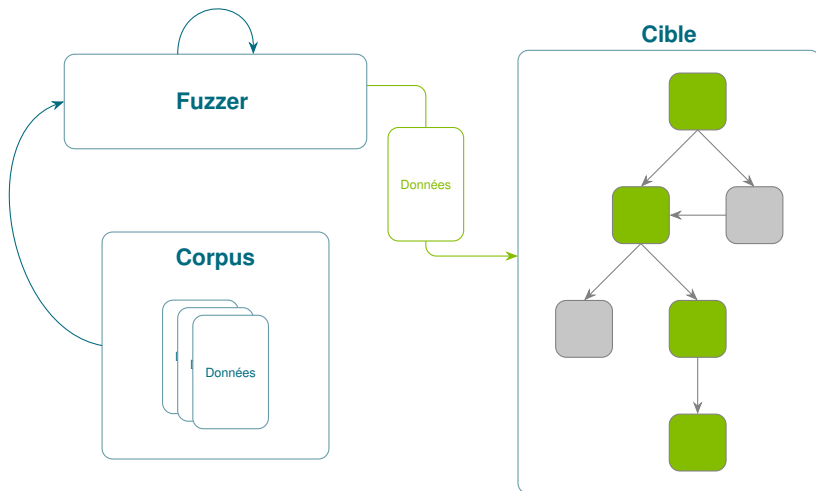
Mesure de la couverture de code

Approche : blocs élémentaires



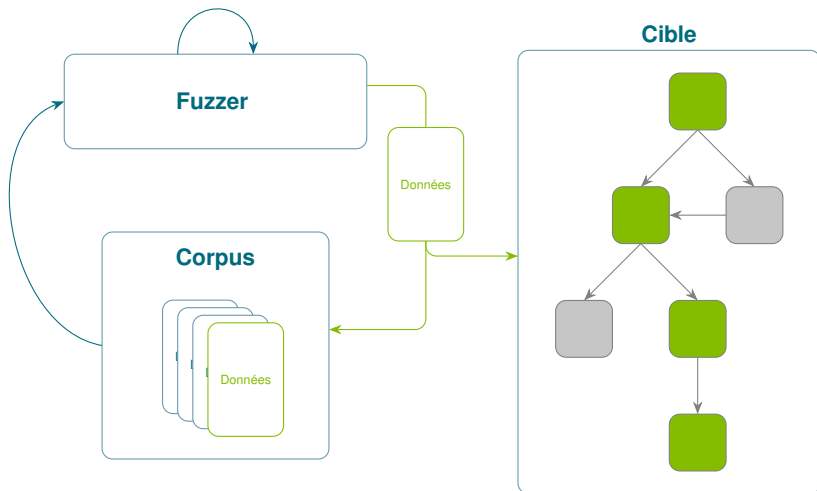
Mesure de la couverture de code

Approche : blocs élémentaires



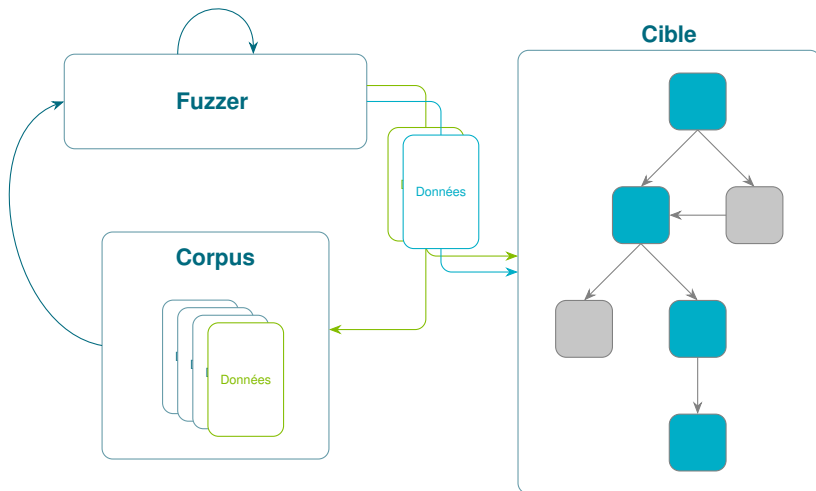
Mesure de la couverture de code

Approche : blocs élémentaires



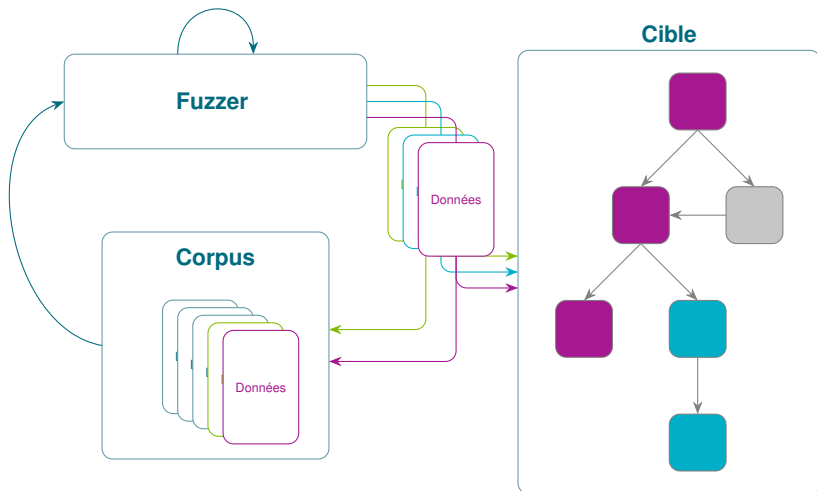
Mesure de la couverture de code

Approche : blocs élémentaires



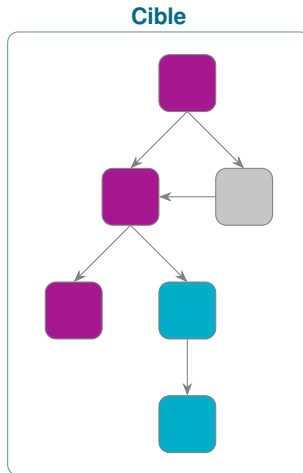
Mesure de la couverture de code

Approche : blocs élémentaires



Instrumentation de la cible

Instrumentation interne à AFL++

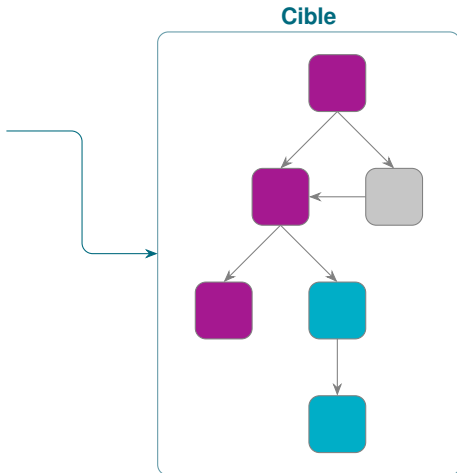


Instrumentation de la cible

Instrumentation interne à AFL++

Code source disponible

- GCC : afl-gcc
- LLVM : afl-clang



Instrumentation de la cible

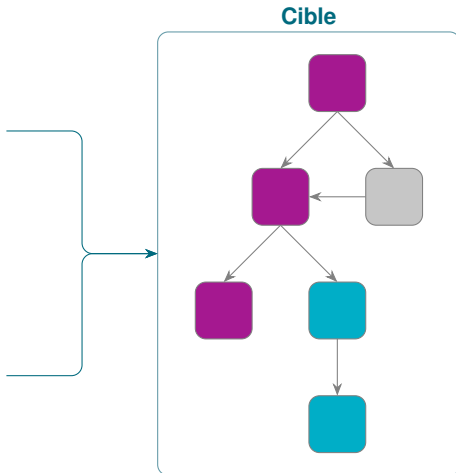
Instrumentation interne à AFL++

Code source disponible

- GCC : afl-gcc
- LLVM : afl-clang

Cible compilée

- Linux : QEMU / Frida
- Windows : Wine
- Embarqué : Unicorn



Instrumentation de la cible

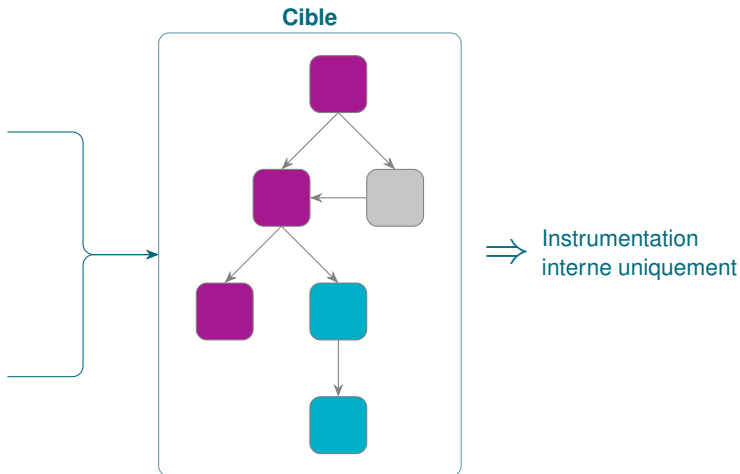
Instrumentation interne à AFL++

Code source disponible

- GCC : afl-gcc
- LLVM : afl-clang

Cible compilée

- Linux : QEMU / Frida
- Windows : Wine
- Embarqué : Unicorn



Instrumentation de la cible

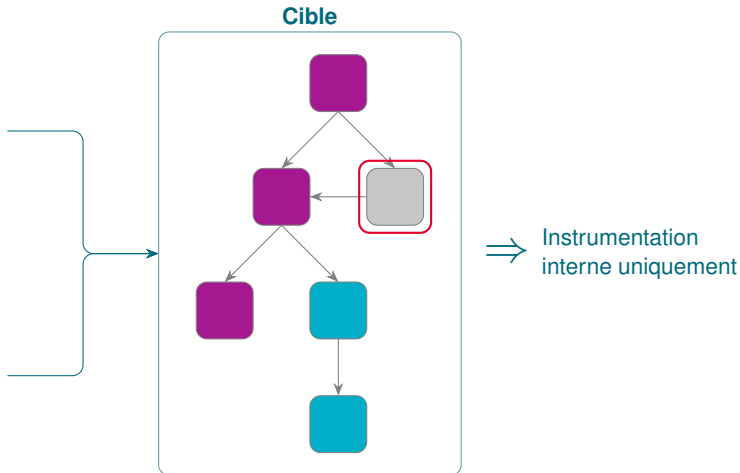
Instrumentation interne à AFL++

Code source disponible

- GCC : afl-gcc
- LLVM : afl-clang

Cible compilée

- Linux : QEMU / Frida
- Windows : Wine
- Embarqué : Unicorn



1 Introduction au fuzzing

2 Couverture de code

3 Présentation de l'outil

4 Conclusion

afl-cov-fast

Une interface unifiée pour obtenir la couverture de code

gcc

```
$ afl-cov-fast.py -m gcc --code-dir src --afl-fuzzing-dir output \  
  --coverage-cmd './a.out @@' -j8
```

LLVM

```
$ afl-cov-fast.py -m llvm --code-dir src --afl-fuzzing-dir output \  
  --coverage-cmd './a.out @@' --binary-path 'a.out' -j8
```

QEMU

```
$ afl-cov-fast.py -m qemu --afl-fuzzing-dir output --afl-path AFLplusplus \  
  --coverage-cmd './a.out @@' -j8
```

Frida

```
$ afl-cov-fast.py -m frida --afl-fuzzing-dir output --afl-path AFLplusplus \  
  --coverage-cmd './a.out @@' -j8
```

État de l'art

Outils de mesure de couverture de code

Code source disponible

- afl-cov
 - Couverture de code par campagne / élément du corpus / fonction
 - Analyse en Python de chaque sortie
 - GCC / LLVM uniquement

État de l'art

Outils de mesure de couverture de code

Code source disponible

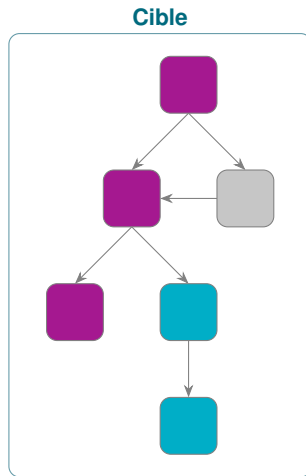
- afl-cov
 - Couverture de code par campagne / élément du corpus / fonction
 - Analyse en Python de chaque sortie
 - GCC / LLVM uniquement

Code source non disponible (QEMU uniquement)

	Version de QEMU	Format de sortie
afl-qemu-cov	Modifiée	CSV
aflq_fast_cov	Modifiée (aflqemu)	Textuel
pyafl_qemu_trace	Originale	JSON

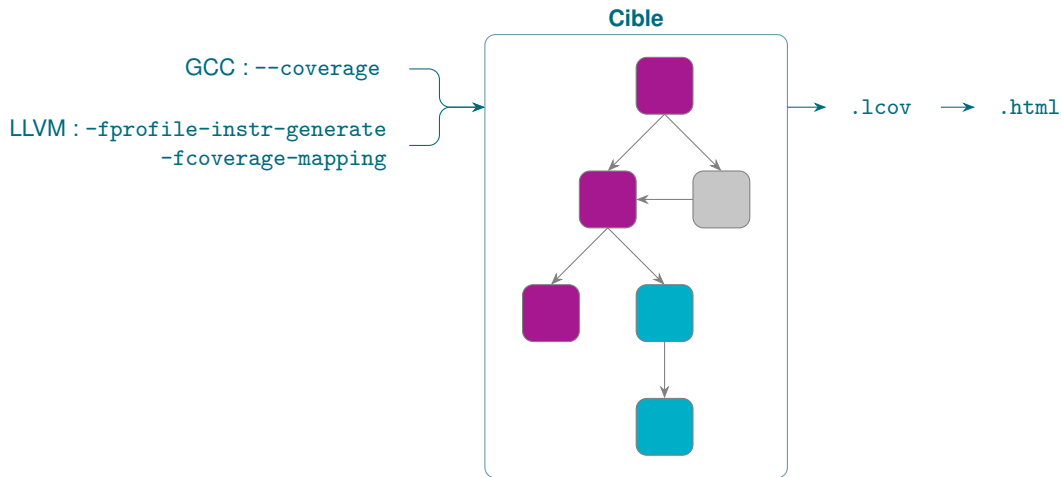
Mesure de la couverture de code

Instrumentation avec code source









Mesure de la couverture de code

Instrumentation avec code source



Visualisation de la couverture de code

afl-cov-fast pour GCC et LLVM

Filename	Line Coverage ↕	Functions ↕
main.c	 79.5 % 58 / 73	50.0 % 1 / 2
x509-cert-parser.c	 51.7 % 1022 / 1976	55.6 % 40 / 72
x509-common.c	 71.5 % 1430 / 2001	76.0 % 76 / 100
x509-crl-parser.c	 84.9 % 823 / 969	100.0 % 33 / 33
x509-parser.c	 81.2 % 26 / 32	100.0 % 2 / 2
x509-utils.c	 100.0 % 8 / 8	100.0 % 1 / 1

```

240 799 : const _pubkey_alg * find_pubkey_alg_by_oid(const u8 *buf, u32 len)
241 : {
242 799 : const _pubkey_alg *found = NULL;
243 799 : const _pubkey_alg *cur = NULL;
244 : u8 k;
245 :
246 799 : if ((buf == NULL) || (len == 0)) {
247 0 : goto out;
248 : }
249 :
250 : /* loop unroll num_known_pubkey_algs;
251 : @ loop invariant 0 <= k <= num_known_pubkey_algs;
252 : @ loop invariant found == NULL;
253 : @ loop assigns cur, found, k;
254 : @ loop variant (num_known_pubkey_algs - k);
255 : */
256 6754 : for (k = 0; k < num_known_pubkey_algs; k++) {
257 : int ret;
258 :
259 6450 : cur = known_pubkey_algs[k];
260 :
261 : /* assert cur == known_pubkey_algs[k]; */
262 6450 : if (cur->alg_der_oid_len != len) {
263 5045 : continue;
264 : }
265 :
266 : /* assert !valid_read(buf + (0 .. (len - 1))); */
267 1405 : ret = !bufs_differ(cur->alg_der_oid, buf, cur->alg_der_oid_len);
268 1405 : if (ret) {
269 495 : found = cur;
270 495 : break;
271 : }
272 : }
273 :
274 304 : out:
275 799 : return found;
276 : }

```

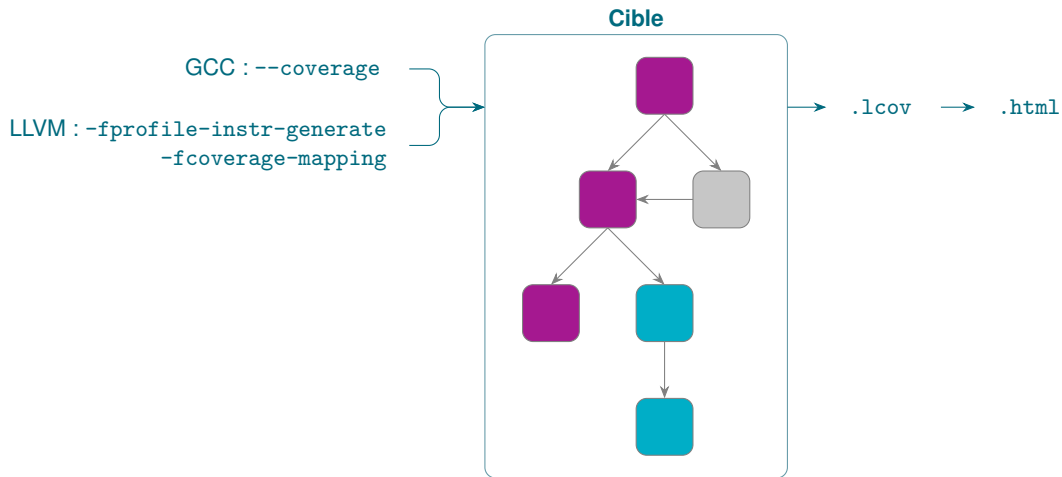
```

523 : /*
524 : @ requires ((u64)off + (u64)len) <= MAX_UINT32;
525 : @ requires ((len > 0) && (cert != NULL)) ==> !valid_read(cert + (off .. (off + len - 1)));
526 : @ requires (raw_pub_off != NULL) ==> !valid(raw_pub_off);
527 : @ requires (raw_pub_len != NULL) ==> !valid(raw_pub_len);
528 : @ requires !separated(cert+{..}, raw_pub_off, raw_pub_len);
529 : @
530 : @ ensures !result < 0 || !result == 0;
531 : @ ensures (len == 0) ==> !result < 0;
532 : @ ensures (cert == NULL) ==> !result < 0;
533 : @
534 : @ assigns *raw_pub_off, *raw_pub_len;
535 : */
536 0 : static int parse_pubkey_eddsa(const u8 *cert, u32 off, u32 len,
537 : u32 exp_pub_len, u32 *raw_pub_off, u32 *raw_pub_len)
538 : {
539 0 : u32 remain, hdr_len = 0, data_len = 0;
540 0 : const u8 *buf = cert + off;
541 : int ret;
542 :
543 0 : if ((cert == NULL) || (len == 0) ||
544 0 : (raw_pub_off == NULL) || (raw_pub_len == NULL)) {
545 0 : ret = -X509_FILE_LINE_NUM_ERR;
546 : ERROR_TRACE_APPEND(X509_FILE_LINE_NUM_ERR);
547 0 : goto out;
548 : }

```

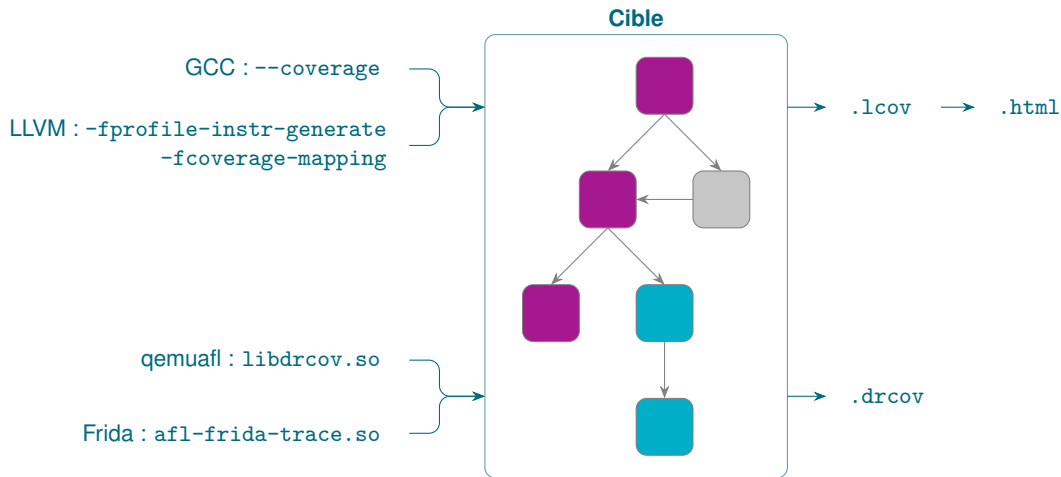
Mesure de la couverture de code

Instrumentation en mode "binaire"



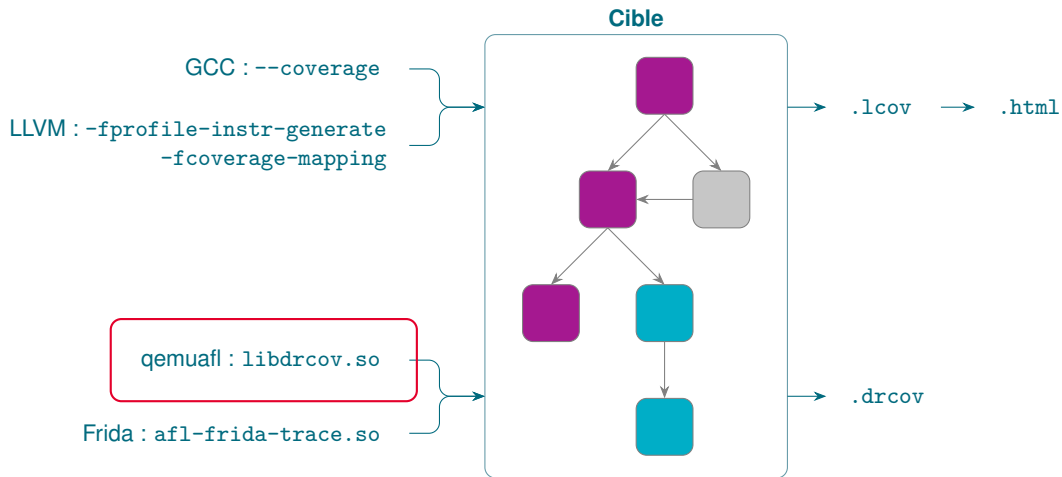
Mesure de la couverture de code

Instrumentation en mode "binaire"



Mesure de la couverture de code

Instrumentation en mode "binaire"



Contributions

Instrumentation avec qemuafI

Contributions

Instrumentation avec qemu afl

Tiny Code Generator (TCG)

- Les plugins TCG QEMU peuvent recevoir des évènements lors de la traduction et l'exécution de blocs
- Le plugin `contrib/plugins/drcov.c` exporte la couverture au format Drcov
 - Non disponible dans qemu afl
 - N'enregistre pas les zones mémoires de la cible lors de l'exécution

Contributions

Instrumentation avec qemu afl

Tiny Code Generator (TCG)

- Les plugins TCG QEMU peuvent recevoir des évènements lors de la traduction et l'exécution de blocs
- Le plugin `contrib/plugins/drcov.c` exporte la couverture au format Drcov
 - Non disponible dans `qemu afl`
 - N'enregistre pas les zones mémoires de la cible lors de l'exécution

Contributions

- Support de `drcov.c` dans `qemu afl` (#56)
- Ajout des zones mémoires de la cible dans l'export (modules chargés, adresses, etc.)
- Support ajouté dans AFL++ (#1956)

Contributions

Instrumentation avec qemu afl

Tiny Code Generator (TCG)

- Les plugins TCG QEMU peuvent recevoir des évènements lors de la traduction et l'exécution de blocs
- Le plugin `contrib/plugins/drcov.c` exporte la couverture au format Drcov
 - Non disponible dans `qemu afl`
 - N'enregistre pas les zones mémoires de la cible lors de l'exécution

Contributions

- Support de `drcov.c` dans `qemu afl` (#56)
- Ajout des zones mémoires de la cible dans l'export (modules chargés, adresses, etc.)
- Support ajouté dans AFL++ (#1956)

Obtention de la couverture

- Utiliser la variable d'environnement `QEMU_PLUGIN` pour charger `libdrcov.so`
- Génère un fichier Drcov à chaque exécution

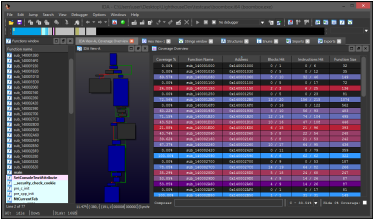
Visualisation de la couverture de code

afl-cov-fast pour QEMU et Frida

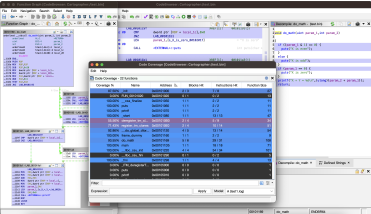
Chargement du fichier Drcov

- Dans IDA ou Binary Ninja avec lighthouse
- Dans Ghidra avec Cartographer ou lightkeeper
- Dans Cutter avec CutterDRcov

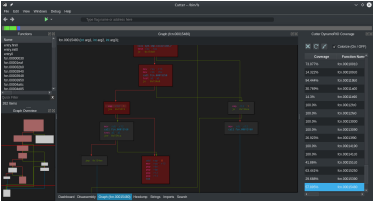
Lighthouse



Cartographer



CutterDRcov



afl-cov-fast

Démo

Démo

Exemple d'une campagne AFL++-QEMU
sur x509-parser

1 Introduction au fuzzing

2 Couverture de code

3 Présentation de l'outil

4 Conclusion

Conclusion

afl-cov-fast

Implémentation

- Script Python utilisant des outils standards (`lcov`, `genhtml`, `afl-qemu-trace`, `drcov-merge`, etc.)
- Support de GCC / LLVM / QEMU / Frida
- Option d'exécution parallèle

Conclusion

afl-cov-fast

Implémentation

- Script Python utilisant des outils standards (`lcov`, `genhtml`, `afl-qemu-trace`, `drcov-merge`, etc.)
- Support de GCC / LLVM / QEMU / Frida
- Option d'exécution parallèle

Utilisation

- 1 Exécuter la campagne
- 2 Récupérer le dossier output de AFL++
- 3 Recompiler la cible pour générer les informations de couverture (si nécessaire)
- 4 Relancer la cible sur chaque entrée
- 5 Combiner les informations de couverture

Conclusion

Merci pour votre attention !

Des questions ?



<https://github.com/airbus-seclab/afl-cov-fast>

—

@AirbusSecLab – <https://airbus-seclab.github.io>