Tendances et contraintes de l'automatisation du fuzzing d'OS embarqué

point de vue d'un industriel

Stéphane Duverger

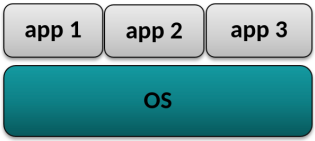
GTSSLR, Paris, 27 Novembre 2019

AIRBUS

Introduction

Particularités

- Noyaux peu dynamiques
- Couches logicielles métier spécifiques
- Importance de la ségrégation spatiale et temporelle



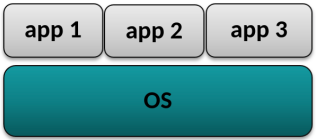
Particularités

- Noyaux peu dynamiques
- Couches logicielles métier spécifiques
- Importance de la ségrégation spatiale et temporelle

Surface d'attaque considérée

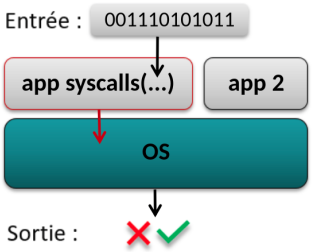
- Depuis une simple application
- Via les appels système

Méthode retenue : *coverage-guided fuzzing*



Coverage-guided fuzzing

- Génération d'entrées conditionnée par le code déjà couvert
- Collecte des informations de couverture de code
- Analyse du comportement de la cible

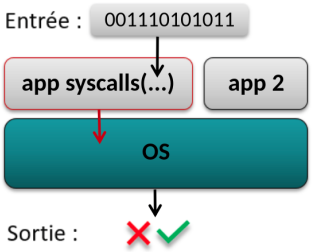


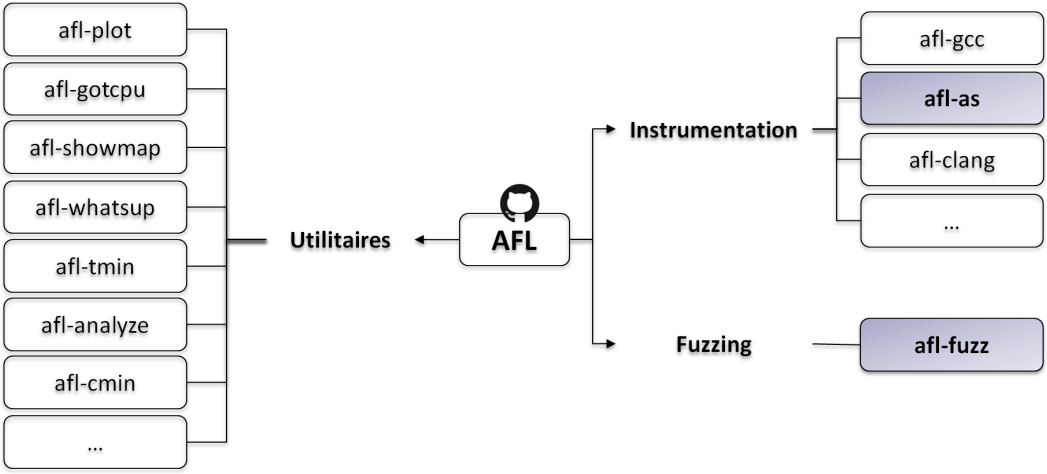
Coverage-guided fuzzing

- Génération d'entrées conditionnée par le code déjà couvert
- Collecte des informations de couverture de code
- Analyse du comportement de la cible

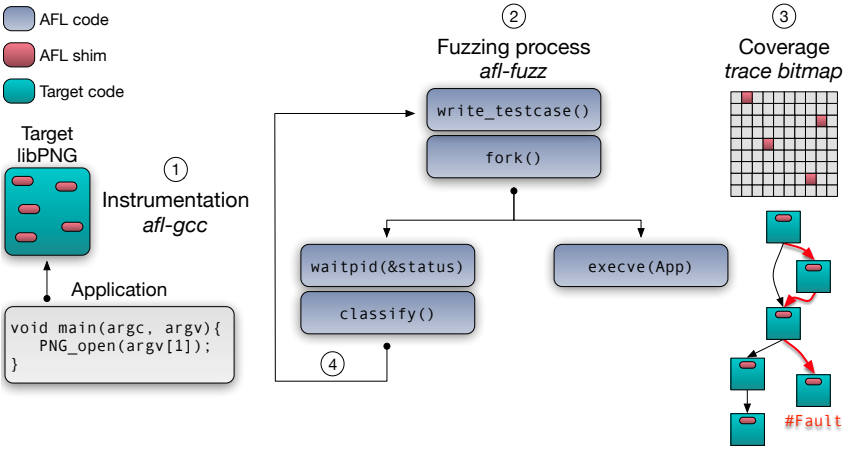
Fuzzer retenu : `AFL`

- Efficace sur de nombreux programmes
- Libre, open-source, livré avec plusieurs utilitaires
- **Notre objectif : Utiliser AFL pour fuzzer les OS**

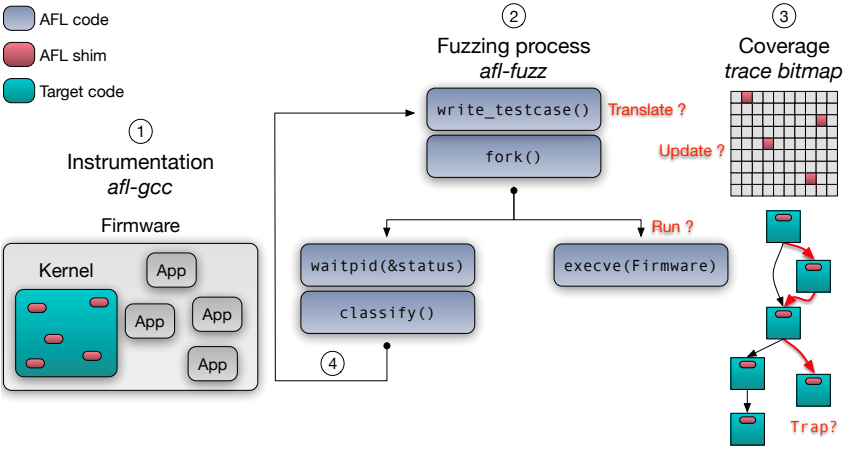



afl-plot
afl-gotcpu
afl-showmap
afl-whatsup
afl-tmin
afl-analyze
afl-cmin
...
Utilitaires
AFL
Instrumentation
afl-gcc
afl-as
afl-clang
...
Fuzzing
afl-fuzz

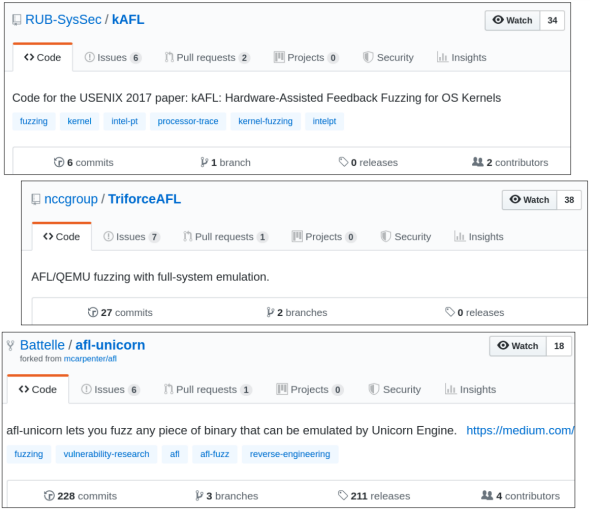
Exemple : *fuzzing* de `libpng`



Exemple : Et pour *fuzzer* un OS ?!



Des candidats ?...

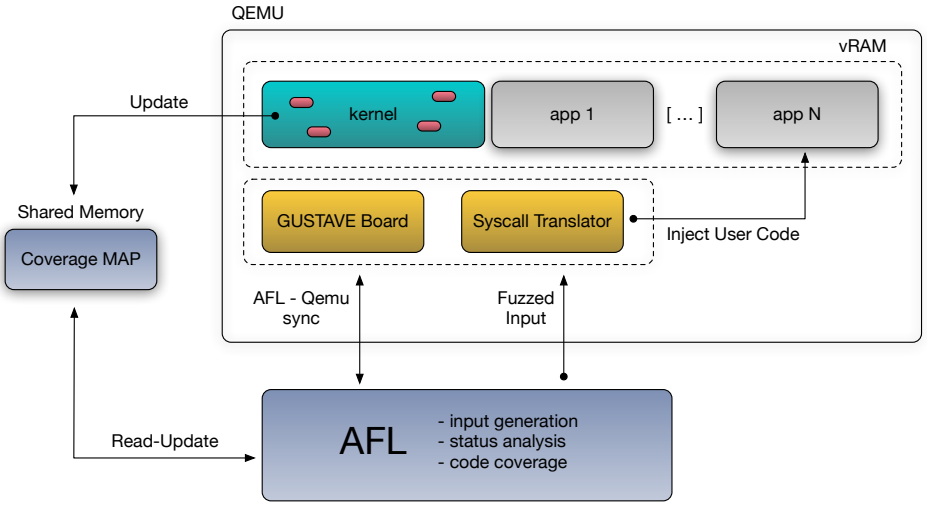


Nos contraintes/souhaits

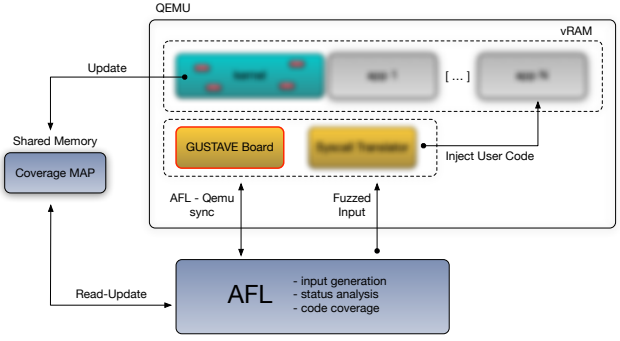
- Indépendant de l'architecture matérielle
- Pas de modification d'`afl-fuzz`
- Pas de dev spécifiques dans l'OS cible
- Outil libre, facilement maintenable

...Conclusion : **"Build your own!" :)**

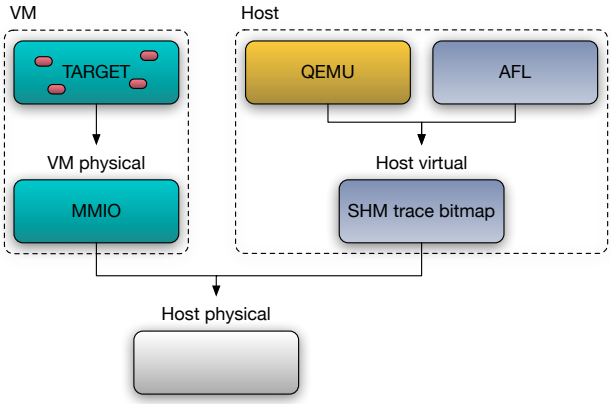
Fonctionnement


QEMU
vRAM
Update
kernel
app 1
[...]
app N
Shared Memory
Coverage MAP
GUSTAVE Board
Syscall Translator
Inject User Code
AFL - Qemu sync
Fuzzed Input
Read-Update
AFL
- input generation
- status analysis
- code coverage

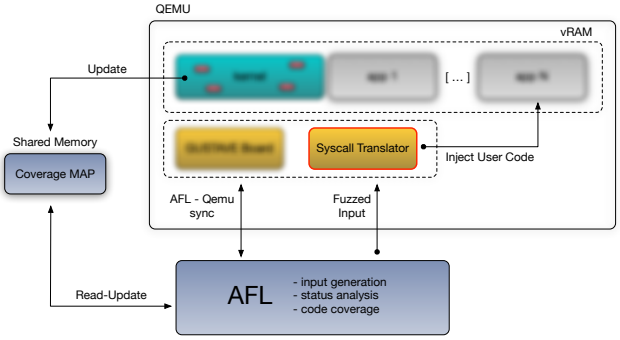
- Implémentation d'une *board* QEMU
  - par architecture matérielle
  - synchronisation avec AFL (*fork-server*)
  - snapshot restauration de la VM
- Pas de modification d'`AFL`
- Pas de modification du TCG
  - instrumentation à la compilation
  - usage de `KVM` possible
  - évite filtrage dynamique



- AFL crée une SHM dans l'hôte
- La cible accède une adresse MMIO arbitraire
- GUSTAVE la redirige vers la bitmap d'AFL
- Aucun surcoût à l'exécution (*like it's app*)



- Transformer les données brutes en programmes
- Séquences d'appels système
- Spécifique à l'architecture et à la cible

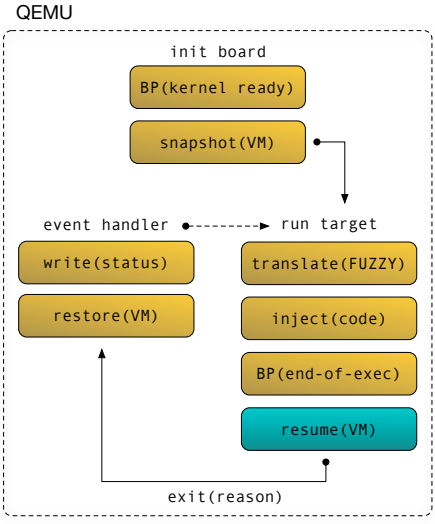


AFL classifie les *test-cases*

- Fin d'exécution normale
- *Time-out*
- Faute (*abort*, *segv*)

GUSTAVE intercepte des évènements

- *Timers* dans QEMU
- *Breakpoints* internes
  - Fin du test injecté
  - Fautes contrôlées : `panic`, `reboot`
- Pas de *véritable* garde-fou noyau
  - Détection d'accès illégitimes *silencieux*
  - Définition d'oracles mémoire



Utilisation

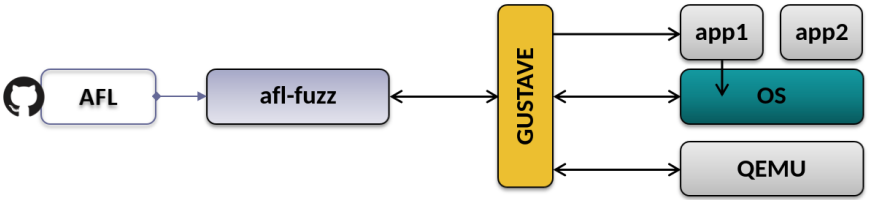
- Instrumentation à la compilation (`afl-gcc`/`afl-as`)
- Optimiser le système selon vos critères :
  - Deux applications basiques, peu de *scheduling*
  - Scénario complexe d'échanges entre applications
  - Focalisé sur un appel système spécifique

```
$ CC=afl-gcc make
[CC] partition: afl-gcc -c -W partition.c -o partition.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 125 locations (32-bit, non-hardened mode, ratio 100%).
```

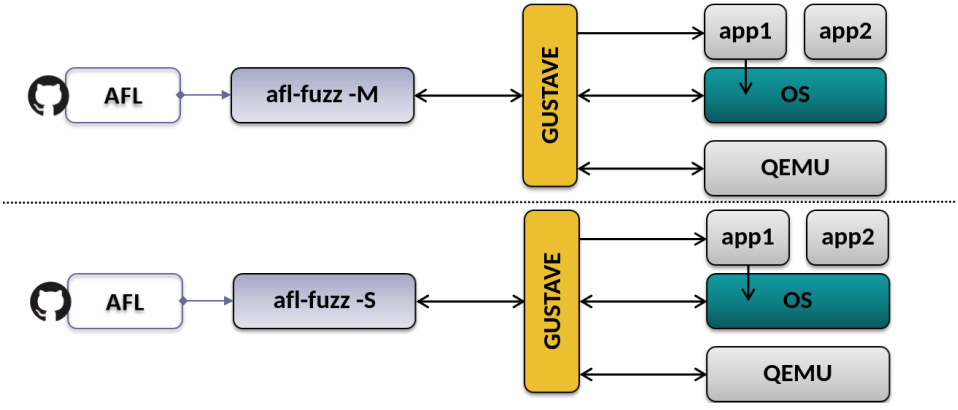
```
$ alias afl="afl-fuzz -d -t 10000 -i /tmp/afl_in \
-o /tmp/afl_out -- qemu-system-ppc -M afl        \
-nographic -bios rom.bin -gustave"

$ afl config/pok_ppc_single.json
```

```
{   "user-timeout": 10000,
    "qemu-overhead": 10,
    "vm-state-template": "/tmp/afl.XXXXXX",
    "afl-control-fd": 198,
    "afl-status-fd": 199,
    "afl-trace-size": 65536,
    "afl-trace-env": "__AFL_SHM_ID",
    "afl-trace-addr": 3758096384,
    "vm-part-base": 221184,
    "vm-part-size": 380768,
    "vm-part-off": 4,
    "vm-nop-size": 65536,
    "vm-fuzz-inj": 221188,
    "vm-size": 0,
    "vm-part-kstack": 0,
    "vm-part-kstack-size": 0,
    "vm-fuzz-ep": 4,
    "vm-fuzz-ep-next": 8,
    "vm-panic": 4293949504,
    "vm-cswitch": 0,
    "vm-cswitch-next": 0
}
```

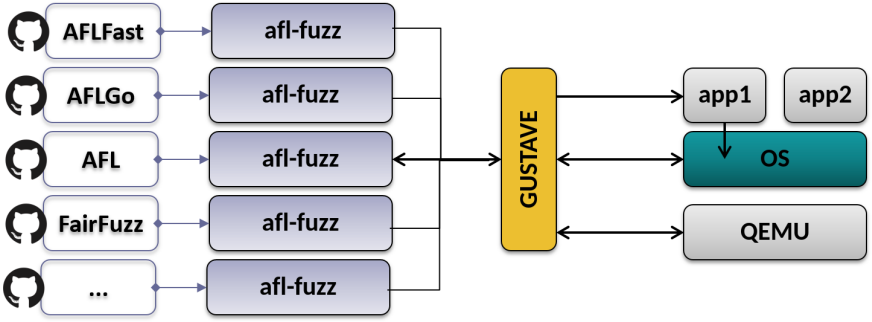
Usage basique



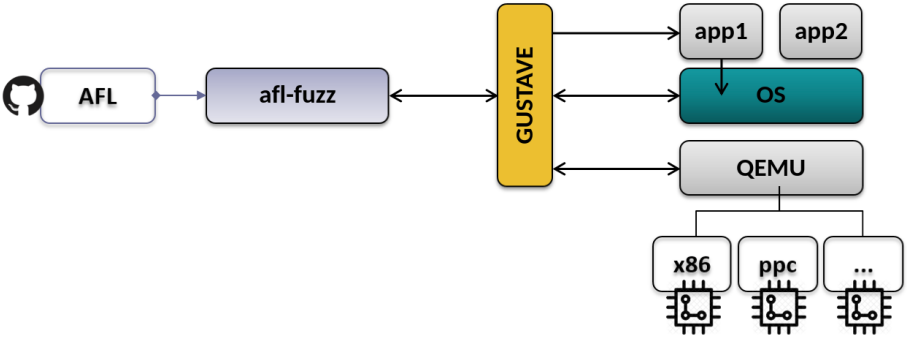
Usage multicœur



Autres dérivés d'afl-fuzz



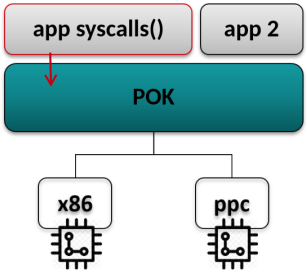
Diverses architectures matérielles



Cas d'usage : POK

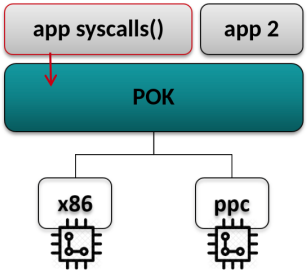
Cible intéressante

- Petit OS, open-source
- Vérifié formellement à 90%
- ... avec des vulnérabilités d'implémentation de ségrégation mémoire :)



Cible intéressante

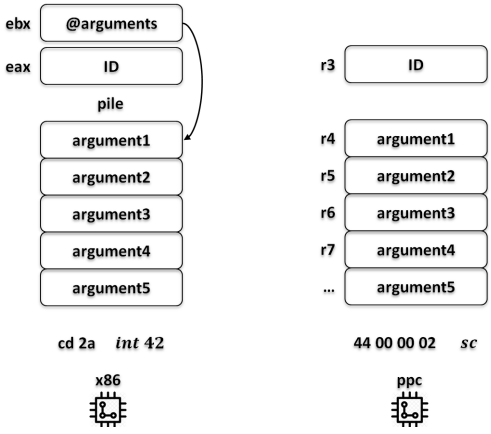
- Petit OS, open-source
- Vérifié formellement à 90%
- ... avec des vulnérabilités d'implémentation de ségrégation mémoire :)



Efforts d'ingénierie

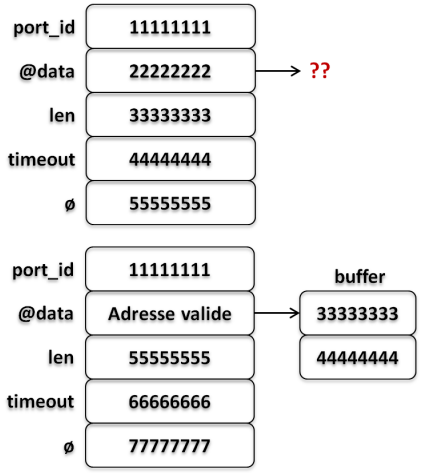
- Analyse
  - Compréhension de la mécanique des appels système / ABI
  - Compréhension de la ségrégation mémoire
- Développements spécifiques
  - De traducteurs d'entrées compatibles à POK
  - D'oracles mémoire correspondant à la logique mémoire de POK

- 50 services noyau exposés aux programmes utilisateur via des appels système
- Logique d'implémentation différente suivant l'architecture matérielle



Diverses possibilités d'interprétation des entrées
(exemple de `POK_SYSCALL_MIDDLEWARE_QUEUEING_SEND`)

- Traitement identique quel que soit le type d'argument
- Traitement spécifique pour les pointeurs de structures
  - Adresse mémoire valide
  - Fuzzing déporté sur le contenu pointé

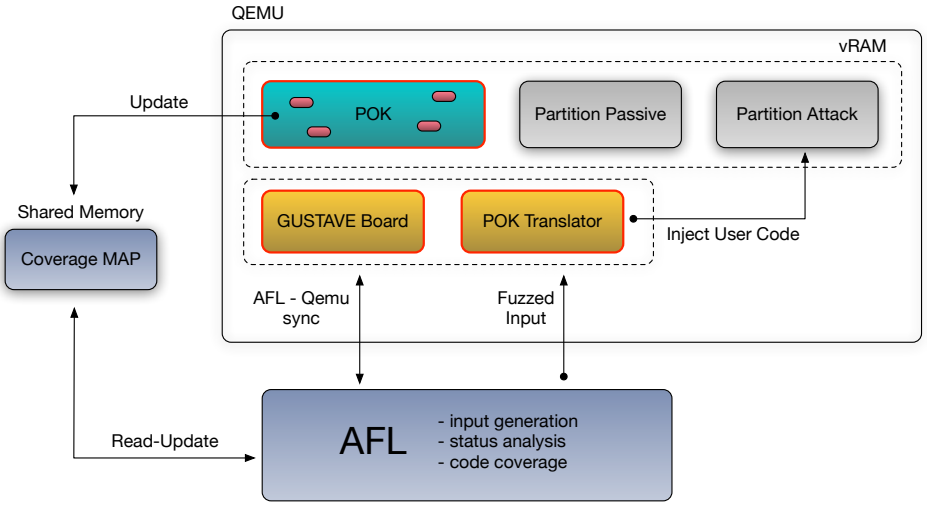


MMU POK

- 1 paire de segments code/données pour chaque programme utilisateur
- 1 paire de segments code/données pour le noyau – en FLAT !! :(
  - Implique une vérification au niveau logiciel

Oracle `GUSTAVE`

- Mapping exclusif des plages mémoire utilisateur / noyau
- Interception des fautes de pages sur les zones non mappées


QEMU
vRAM
Update
POK
Partition Passive
Partition Attack
Shared Memory
Coverage MAP
GUSTAVE Board
POK Translator
Inject User Code
AFL - Qemu sync
Fuzzed Input
Read-Update
AFL
- input generation
- status analysis
- code coverage

Vulnérabilités trouvées

- Vulnérabilité attendue retrouvée
  - Absence de vérification d'une adresse passée en argument
  - Conséquence : possibilité d'écriture arbitraire
- Détection automatique de tous les autres appels système vulnérables au même problème
  - 25 autres possibilités d'écriture arbitraire

Performances

- environ 500 tests/sec
- Stabilité (déterminisme) du fuzzing proche de 100%

Retour d'expérience

- Instrumentation à la compilation
  - préparation firmware spécifique
  - forge complexe, figée, *Dev-lol-Ops*
  - plus compliqué que prévu

Vers l'instrumentation binaire

- Instrumentation à la compilation
  - préparation firmware spécifique
  - forge complexe, figée, *Dev-lol-Ops*
  - plus compliqué que prévu
- Instrumentation binaire
  - modification du TCG
  - plus simple que prévu
  - fenêtre ouverte vers de l'optimisation

- Instrumentation à la compilation
  - préparation firmware spécifique
  - forge complexe, figée, *Dev-lol-Ops*
  - plus compliqué que prévu
- Instrumentation binaire
  - modification du TCG
  - plus simple que prévu
  - fenêtre ouverte vers de l'optimisation
- Tirer profit d'évolutions récentes
  - `https://abiondo.me/2018/09/21/improving-afl-qemu-mode/`
  - `https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html`

- Tiny Code Generator de QEMU
- historiquement *QOP code generator* de Paul Brook
- Guest vers IR vers Host
- décomposition du code simulé en *Translation Blocks*
- nombreuses optimisations (*chaining*, *helpers*)

```
static void * qemu_tcg_rr_cpu_thread_fn(void *arg)
{
  while (1)
     if (cpu_can_run(cpu))
        r = cpu_exec(cpu);
}

int cpu_exec(CPUState *cpu)
{
  while (!cpu_handle_exception(cpu, &ret))
     while (!cpu_handle_interrupt(cpu, &last_tb)) {
        tb = tb_find(cpu, last_tb, tb_exit, cflags);
        cpu_loop_exec_tb(cpu, tb, &last_tb, &tb_exit);
     }
}
```

```
TranslationBlock *tb_find(CPUState *cpu,
                          TranslationBlock *last_tb,
                          int tb_exit, uint32_t cf_mask)
{
  tb = tb_lookup__cpu_state(cpu, &pc, &cs_base,
                            &flags, cf_mask);
  if (tb == NULL)
      tb = tb_gen_code(cpu, pc, cs_base,
                       flags, cf_mask);
}

TranslationBlock *tb_gen_code(CPUState *cpu,
                              target_ulong pc,
                              target_ulong cs_base,
                              uint32_t flags, int cflags)
{
    tb = tb_alloc(pc);
    gen_intermediate_code(cpu, tb, max_insns);

    /* generate machine code */
    gen_code_size = tcg_gen_code(tcg_ctx, tb);
}
```

- PowerPC 32bits

```
fff00100:  lis   r1,1

fff00104:  ori   r1,r1,0x409c

fff00108:  xor   r0,r0,r0

fff0010c:  stw   r0,4(r1)

fff00110:  mtmsr  r0
```

- TCG IR

```
fff00100:  movi_i32     r1,$0x10000

fff00104:  movi_i32     tmp0,$0x409c
           or_i32       r1,r1,tmp0

fff00108:  movi_i32     r0,$0x0

fff0010c:  movi_i32     tmp1,$0x4
           add_i32      tmp0,r1,tmp1
           qemu_st_i32  r0,tmp0,beul,3

fff00110:  movi_i32     nip,$0xfff00114
           mov_i32      tmp0,r0
           call         store_msr,$0,tmp0
           movi_i32     nip,$0xfff00114
           exit_tb      $0x0
           set_label    $L0
           exit_tb      $0x7f5a0caf8043
```

- Intel x86 64bits

```
.7f5a0caf810b:  movl    $0x1409c, 4(%rbp)
.7f5a0caf8112:  xorl    %ebx, %ebx
.7f5a0caf8114:  movl    %ebx, (%rbp)
.7f5a0caf8117:  movl    $0x140a0, %r12d
.7f5a0caf811d:  movl    %r12d, %edi
.7f5a0caf8129:  addq    0x398(%rbp), %rdi
...
.7f5a0caf8159:  movq    %rbp, %rdi
.7f5a0caf815c:  movl    %ebx, %esi
.7f5a0caf815e:  callq   *0x34(%rip)
.7f5a0caf8164:  movl    $0xfff00114, 0x16c(%rbp)
.7f5a0caf8182:  movl    %ebx, %edx
.7f5a0caf8184:  movl    $0xa3, %ecx
.7f5a0caf8189:  leaq    -0x41(%rip), %r8
.7f5a0caf8190:  pushq   %r8
.7f5a0caf8192:  jmpq    *8(%rip)
.7f5a0caf8198:  .quad   0x000055d62e46eba0
.7f5a0caf81a0:  .quad   0x000055d62e3895a0
```

- PowerPC 32bits

```
fff00100:  lis   r1,1

fff00104:  ori   r1,r1,0x409c

fff00108:  xor   r0,r0,r0

fff0010c:  stw   r0,4(r1)


fff00110:  mtmsr  r0
```

- TCG IR

```
fff00100:  movi_i32     r1,$0x10000

fff00104:  movi_i32     tmp0,$0x409c
           or_i32       r1,r1,tmp0

fff00108:  movi_i32     r0,$0x0

fff0010c:  movi_i32     tmp1,$0x4
           add_i32      tmp0,r1,tmp1
           qemu_st_i32  r0,tmp0,beul,3

fff00110:  movi_i32     nip,$0xfff00114
           mov_i32      tmp0,r0
           call         store_msr,$0,tmp0
           movi_i32     nip,$0xfff00114
           exit_tb      $0x0
           set_label    $L0
           exit_tb      $0x7f5a0caf8043
```

- Intel x86 64bits

```
.7f5a0caf810b:  movl    $0x1409c, 4(%rbp)
.7f5a0caf8112:  xorl    %ebx, %ebx
.7f5a0caf8114:  movl    %ebx, (%rbp)
.7f5a0caf8117:  movl    $0x140a0, %r12d
.7f5a0caf811d:  movl    %r12d, %edi
.7f5a0caf8129:  addq    0x398(%rbp), %rdi
...
.7f5a0caf8159:  movq    %rbp, %rdi
.7f5a0caf815c:  movl    %ebx, %esi
.7f5a0caf815e:  callq   *0x34(%rip)
.7f5a0caf8164:  movl    $0xfff00114, 0x16c(%rbp)
.7f5a0caf8182:  movl    %ebx, %edx
.7f5a0caf8184:  movl    $0xa3, %ecx
.7f5a0caf8189:  leaq    -0x41(%rip), %r8
.7f5a0caf8190:  pushq   %r8
.7f5a0caf8192:  jmpq    *8(%rip)
.7f5a0caf8198:  .quad   0x000055d62e46eba0
.7f5a0caf81a0:  .quad   0x000055d62e3895a0
```

- TCG IR

```
call    store_msr,$0,tmp0
```

- Intel x86 64bits

```
.7f5a0caf815e:  callq    *0x34(%rip)
...
.7f5a0caf8198:  .quad  0x000055d62e46eba0
```

- TCG IR

```
call    store_msr,$0,tmp0
```

- TCG helpers
  - certaines insn Guest trop complexes
  - implémentation via des *helpers* en C
  - compilé avec QEMU, génère un `call` natif
  - similaire `hypercall/paravirt_ops`
- Block Chaining
  - éviter de revenir dans QEMU pour `next TB`
  - fixe offset directement dans les TBs

- Intel x86 64bits

```
.7f5a0caf815e:  callq    *0x34(%rip)
...
.7f5a0caf8198:  .quad  0x000055d62e46eba0
```

```
(gdb) x/i 0x000055d62e46eba0
   0x000055d62e46eba0 <helper_store_msr>: push %r12
```

https://github.com/nccgroup/TriforceAFL

- Fuzzer d'OS fondé sur AFL et QEMU
  - adaptation du *qemu-mode* (system)
  - trace dans QEMU après chaque exec(TB)
  - insn dans le Guest pour synchro AFL

```
tcg_target_ulong cpu_tb_exec(CPUState *cpu, uint8_t *tb_ptr)
{
  CPUArchState *env = cpu->env_ptr;
  uintptr_t next_tb;

  cpu->can_do_io = 0;
  target_ulong pc = env->eip;
  next_tb = tcg_qemu_tb_exec(env, tb_ptr);
  cpu->can_do_io = 1;

  /* we executed it, trace it */
  AFL_QEMU_CPU_SNIPPET2(env, pc);
}
```

https://github.com/nccgroup/TriforceAFL

- Fuzzer d'OS fondé sur AFL et QEMU
  - adaptation du *qemu-mode* (system)
  - trace dans QEMU après chaque exec(TB)
  - insn dans le Guest pour synchro AFL
- Mauvais design (historique qemu-mode)
  - victime du *block chaining*
  - impacte la stabilité dans AFL
  - impacte les performances
  - orienté Linux, module kernel, loader ...

```
tcg_target_ulong cpu_tb_exec(CPUState *cpu, uint8_t *tb_ptr)
{
  CPUArchState *env = cpu->env_ptr;
  uintptr_t next_tb;

  cpu->can_do_io = 0;
  target_ulong pc = env->eip;
  next_tb = tcg_qemu_tb_exec(env, tb_ptr);
  cpu->can_do_io = 1;

  /* we executed it, trace it */
  AFL_QEMU_CPU_SNIPPET2(env, pc);
}
```

La bonne approche (1/2)

https://abiondo.me/2018/09/21/improving-afl-qemu-mode

```
TranslationBlock *tb_gen_code(CPUState *cpu, ..., int cflags)
{
  tb = tb_alloc(pc);
  tcg_func_start(tcg_ctx);

  afl_gen_trace(pc);

  gen_intermediate_code(cpu, tb, max_insns);
  /* generate machine code */
  gen_code_size = tcg_gen_code(tcg_ctx, tb);
}
```

```
static void afl_gen_trace(target_ulong cur_loc)
{
  /* index = prev_loc ^ cur_loc */
  prev_loc_ptr = tcg_const_ptr(&prev_loc);
  index = tcg_temp_new();
  tcg_gen_ld_tl(index, prev_loc_ptr, 0);
  tcg_gen_xori_tl(index, index, cur_loc);

  /* afl_area_ptr[index]++ */
  count_ptr = tcg_const_ptr(afl_area_ptr);
  tcg_gen_add_ptr(count_ptr, count_ptr, TCGV_NAT_TO_PTR(index));
  count = tcg_temp_new();
  tcg_gen_ld8u_tl(count, count_ptr, 0);
  tcg_gen_addi_tl(count, count, 1);
  tcg_gen_st8_tl(count, count_ptr, 0);

  /* prev_loc = cur_loc >> 1 */
  new_prev_loc = tcg_const_tl(cur_loc >> 1);
  tcg_gen_st_tl(new_prev_loc, prev_loc_ptr, 0);
}
```

La bonne approche (1/2)

https://abiondo.me/2018/09/21/improving-afl-qemu-mode

```
TranslationBlock *tb_gen_code(CPUState *cpu, ..., int cflags)
{
  tb = tb_alloc(pc);
  tcg_func_start(tcg_ctx);

  afl_gen_trace(pc);

  gen_intermediate_code(cpu, tb, max_insns);
  /* generate machine code */
  gen_code_size = tcg_gen_code(tcg_ctx, tb);
}
```

- Andrea Biondo
  - générer en IR le *code coverage*
  - durant la génération des TBs
  - intéressant, mais ...

```
static void afl_gen_trace(target_ulong cur_loc)
{
  /* index = prev_loc ^ cur_loc */
  prev_loc_ptr = tcg_const_ptr(&prev_loc);
  index = tcg_temp_new();
  tcg_gen_ld_tl(index, prev_loc_ptr, 0);
  tcg_gen_xori_tl(index, index, cur_loc);

  /* afl_area_ptr[index]++ */
  count_ptr = tcg_const_ptr(afl_area_ptr);
  tcg_gen_add_ptr(count_ptr, count_ptr, TCGV_NAT_TO_PTR(index));
  count = tcg_temp_new();
  tcg_gen_ld8u_tl(count, count_ptr, 0);
  tcg_gen_addi_tl(count, count, 1);
  tcg_gen_st8_tl(count, count_ptr, 0);

  /* prev_loc = cur_loc >> 1 */
  new_prev_loc = tcg_const_tl(cur_loc >> 1);
  tcg_gen_st_tl(new_prev_loc, prev_loc_ptr, 0);
}
```

La bonne approche (2/2)

https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html

```
TranslationBlock *tb_gen_code(CPUState *cpu, ..., int cflags)
{
  tb = tb_alloc(pc);
  tcg_func_start(tcg_ctx);

  afl_gen_trace(pc);

  gen_intermediate_code(cpu, tb, max_insns);
  /* generate machine code */
  gen_code_size = tcg_gen_code(tcg_ctx, tb);
}
```

```
void afl_maybe_log(target_ulong cur_loc)
{
  register uintptr_t afl_idx = cur_loc ^ __global_afl->prev_loc;

  ((uint8_t*)__global_afl->trace_bits)[afl_idx]++;
  __global_afl->prev_loc = cur_loc >> 1;
}

void afl_gen_trace(target_ulong cur_loc)
{
  if (!__global_afl->ready || !afl_is_kernel(&__global_afl->arch))
      return;

  cur_loc = (cur_loc >> 4) ^ (cur_loc << 8);
  cur_loc &= __global_afl->config.afl.trace_size - 1;

  tcg_gen_afl_maybe_log_call(cur_loc);
}
```

La bonne approche (2/2)

https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html

```
TranslationBlock *tb_gen_code(CPUState *cpu, ..., int cflags)
{
  tb = tb_alloc(pc);
  tcg_func_start(tcg_ctx);

  afl_gen_trace(pc);

  gen_intermediate_code(cpu, tb, max_insns);
  /* generate machine code */
  gen_code_size = tcg_gen_code(tcg_ctx, tb);
}
```

- Andrea Fioraldi (pour AFLplusplus)
  - réflexions récentes (07/2019)
  - appeler un *helper* pour le coverage
  - notre implémentation actuelle !

```
void afl_maybe_log(target_ulong cur_loc)
{
  register uintptr_t afl_idx = cur_loc ^ __global_afl->prev_loc;

  ((uint8_t*)__global_afl->trace_bits)[afl_idx]++;
  __global_afl->prev_loc = cur_loc >> 1;
}

void afl_gen_trace(target_ulong cur_loc)
{
  if (!__global_afl->ready || !afl_is_kernel(&__global_afl->arch))
      return;

  cur_loc = (cur_loc >> 4) ^ (cur_loc << 8);
  cur_loc &= __global_afl->config.afl.trace_size - 1;

  tcg_gen_afl_maybe_log_call(cur_loc);
}
```

- TCG IR

```
movi_i32     tmp0,$0xad3a
call         (null),$0,tmp0

ld_i32       tmp1,env,$0xffffffd8
movi_i32     tmp2,$0x0
brcond_i32   tmp1,tmp2,lt,$L0
movi_i32     tmp2,$0x14
add_i32      tmp1,r1,tmp2
```

- Intel x86 64bits

```
.7f152f0c8580:  movl    $0xad3a, %edi
.7f152f0c8585:  callq   *0x95(%rip)
.7f152f0c858b:  movl    -0x28(%rbp), %ebx
...
.7f152f0c85fd:  movl    $0xa3, %edx
.7f152f0c8602:  leaq    -0x40(%rip), %rcx
.7f152f0c8609:  callq   *9(%rip)
.7f152f0c860f:  movl    %eax, %ebx
.7f152f0c8611:  jmp     0x7f152f0c85c9
.7f152f0c8618:  .quad   0x0000559a061d9070
.7f152f0c8620:  .quad   0x0000559a0620b410
```

```
(gdb) x/i 0x0000559a0620b410
   0x559a0620b410 <afl_maybe_log>: mov 0xaa3e49(%rip),%rdx
```

- TCG IR

```
movi_i32     tmp0,$0xad3a
call         (null),$0,tmp0

ld_i32       tmp1,env,$0xffffffd8
movi_i32     tmp2,$0x0
brcond_i32   tmp1,tmp2,lt,$L0
movi_i32     tmp2,$0x14
add_i32      tmp1,r1,tmp2
```

- similaire au *source level instrumentation*
- génération d'un index par TB
- appel à la fonction de *coverage*
- qui met à jour la *trace bitmap*
- performances identiques

- Intel x86 64bits

```
.7f152f0c8580:  movl    $0xad3a, %edi
.7f152f0c8585:  callq   *0x95(%rip)
.7f152f0c858b:  movl    -0x28(%rbp), %ebx
...
.7f152f0c85fd:  movl    $0xa3, %edx
.7f152f0c8602:  leaq    -0x40(%rip), %rcx
.7f152f0c8609:  callq   *9(%rip)
.7f152f0c860f:  movl    %eax, %ebx
.7f152f0c8611:  jmp     0x7f152f0c85c9
.7f152f0c8618:  .quad   0x0000559a061d9070
.7f152f0c8620:  .quad   0x0000559a0620b410
```

```
(gdb) x/i 0x0000559a0620b410
   0x559a0620b410 <afl_maybe_log>: mov 0xaa3e49(%rip),%rdx
```

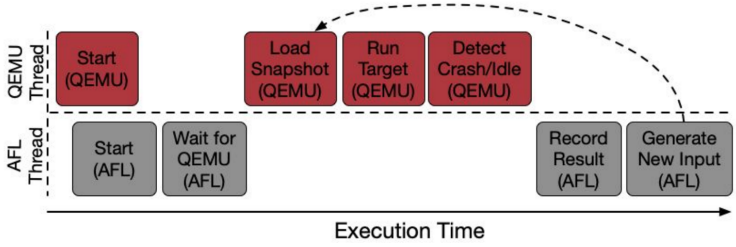
- kAFL : virtualisation **x86** + Intel PT (Processor Trace)
  `https://github.com/RUB-SysSec/kAFL`

- kAFL : virtualisation **x86** + Intel PT (Processor Trace)
  `https://github.com/RUB-SysSec/kAFL`
- PowerFL: VxWorks + AFL + QEMU (assez proche de GUSTAVE)
  `https://www.petergoodman.me/docs/qpss-2019-slides.pdf`

PowerFL = AFL + QEMU + VxWorks

TRAIL OF BITS

- **PowerFL can fuzz across architecture and OS boundaries**
  - Novel solutions for i/o passthrough, crash/idle detection, device hooks.



- Travaux de Brandon Falk (`@gamozolabs`)
  - Vectorized Emulation: Pwnie Award *most innovative research*
    `https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html`
  - Byte Oriented MMU
    `https://gamozolabs.github.io/fuzzing/2018/11/19/vectorized_emulation_mmu.html`
  - insn SIMD pour paralléliser les *test-cases*
  - AVX-512 JiT compiler
  - $\sim$ 100 billion emulated instructions per second
  - $>$ 1 million fuzz cases per second

Pistes de recherche (1/2)

- Repenser les limitations en plongeant dans le TCG
  - frontière émulation - virtualisation moins nette
  - futures optimisations
    - conserver cache du TCG pour le code kernel
    - modifier le système de snapshots de QEMU
    - fonctionnalités du CPU host (Intel PT)
  - généricité de l'injection des syscalls
    - description model haut-niveau pour la cible
    - implémentation générique dans QEMU en x86-64
    - *paravirt* dans les TBs vers le handler kernel cible

Pistes de recherche (1/2)

- Repenser les limitations en plongeant dans le TCG
  - frontière émulation - virtualisation moins nette
  - futures optimisations
    - conserver cache du TCG pour le code kernel
    - modifier le système de snapshots de QEMU
    - fonctionnalités du CPU host (Intel PT)
  - généricité de l'injection des syscalls
    - description model haut-niveau pour la cible
    - implémentation générique dans QEMU en x86-64
    - *paravirt* dans les TBs vers le handler kernel cible
- Bénéfice directe des optimisations
  - finesse d'instrumentation
  - oracles mémoire de B. Falk
    - granularité à l'octet
    - monitorer des buffers critiques
    - RAW (read-after-write)

Pistes de recherche (2/2)

- Interfaçage à d'autres fuzzer/stratégies
  - *redqueen* et l'*input-to-state correspondence*
    `https://github.com/RUB-SysSec/redqueen`
  - *syzkaller* et son parser de headers
    `https://github.com/google/syzkaller`
  - *honggfuzz*
    `https://google.github.io/honggfuzz`

Conclusion

- AFL peut fuzzer des OS embarqués (*like it's app*)
- Intégration dans des *boards* x86 et PPC
- Conditionné par :
  - Support de l'OS dans QEMU
  - Compréhension de l'ABI/stratégie de ségrégation mémoire
- Questions existentielles:
  - Course à l'optimisation
  - Pertinence de la simulation système complet

Merci pour votre attention :)
Questions ?

stephane.duverger@airbus.com

@AirbusSecLab
https://github.com/airbus-seclab/gustave