Gustave-NG: Fuzz It Like It's App

(feat. QEMU and AFL)

A. Gantet, S. Duverger

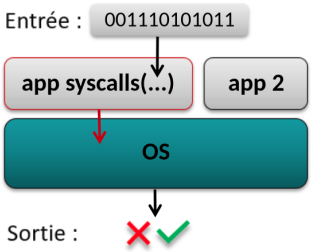
March 3, 2021

Coverage-guided fuzzing

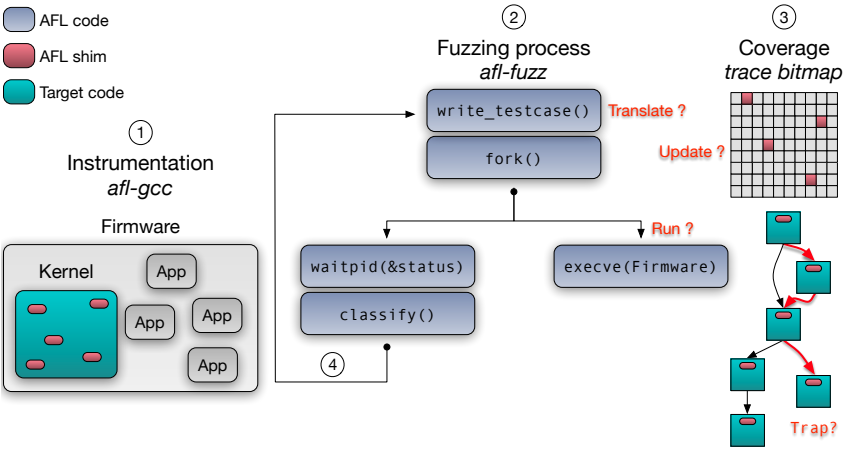
- collect code coverage stats
- input generation based on actual coverage
- target behavior analysis

Candidate : `AFL`

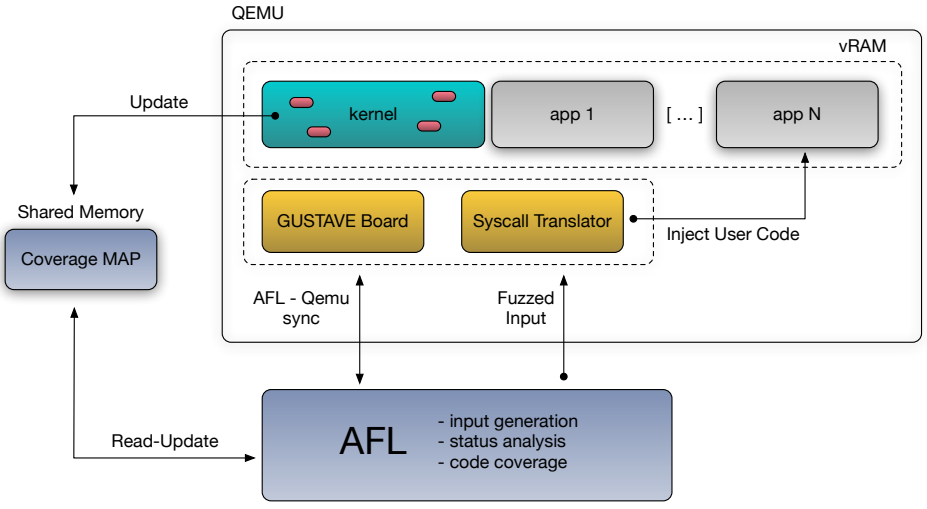
- success on a lot of softwares (apps, libs, ...)
- free, open-source, bundled with side tools
- **Goal : use AFL to fuzz OS kernels**



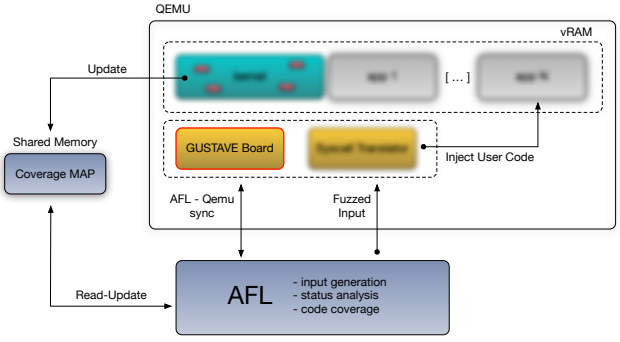
How to fuzz an OS kernel ?!



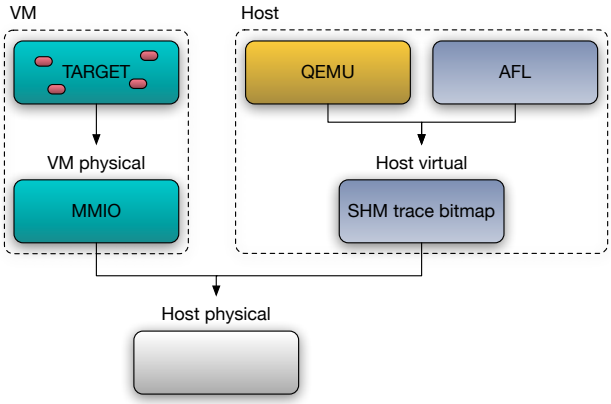
Internals


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

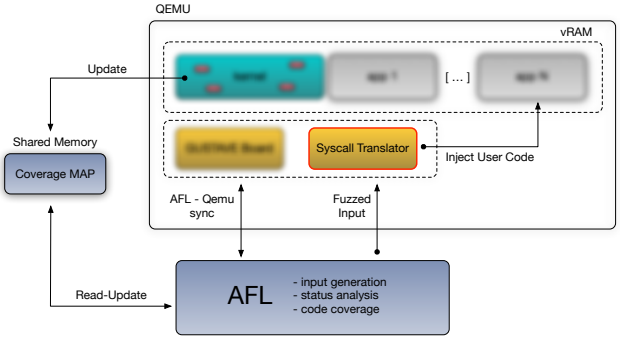
- Implement a new QEMU board
  - for each architecture
  - AFL sync(fork-server)
  - VM restore snapshot
- untouched, original AFL
- no TCG modification/hacks
  - target build-time instrumentation
  - prevent dynamic filtering (user/kernel, specific parts only)



- AFL has a SHM in Host memory
- target writes it through arbitrary MMIO address
- GUSTAVE redirect this MMIO to AFLbitmap
- no execution overhead (like it's app)



- translated raw inputs into programs
- sequence of system calls
- target OS/architecture specific

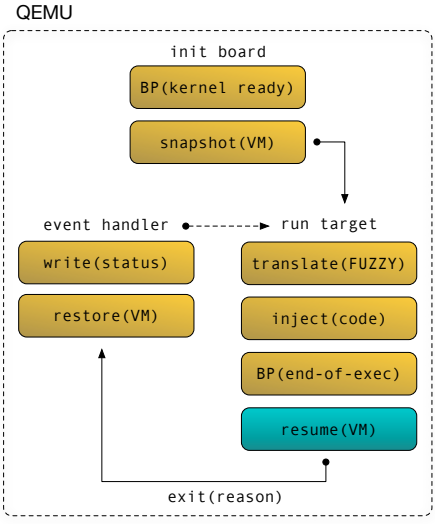


AFL classifies test-cases

- normal exit
- Time-out
- Abort, SegV

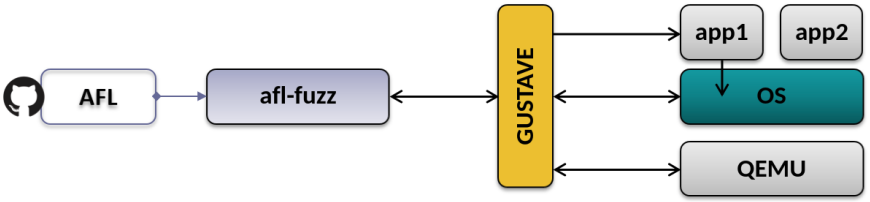
GUSTAVE intercepts events

- QEMU Timers
- internal Breakpoints
  - end of injected test case code
  - known faulty locations : `panic`, `reboot`
- No easy way to:
  - detect illegitimate accesses
  - need to trick memory configuration

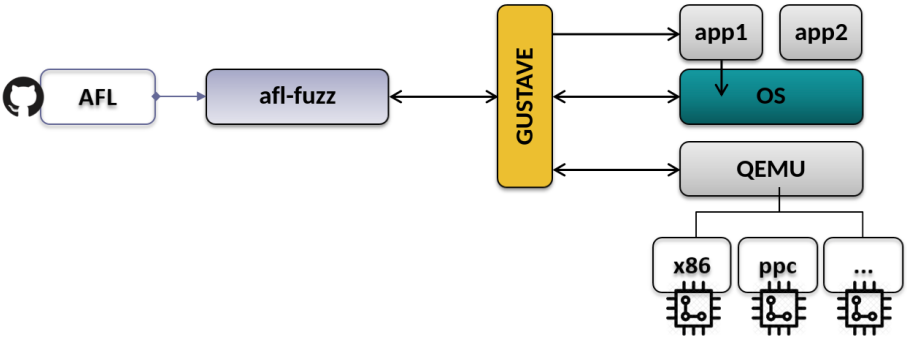


Usage

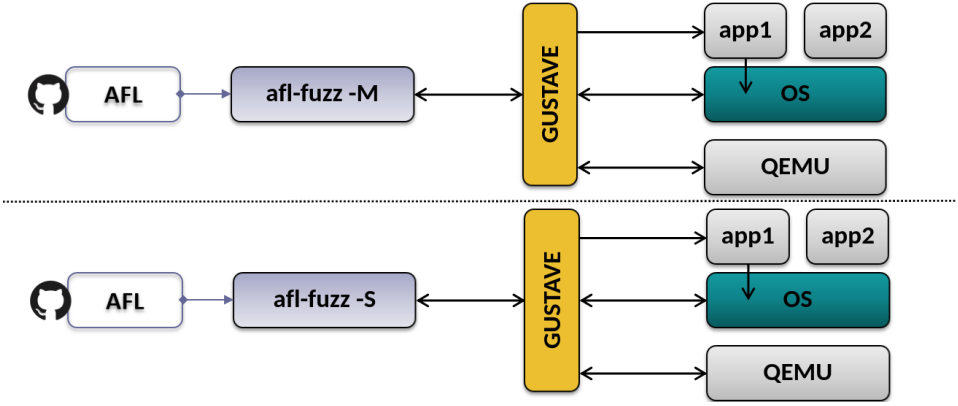
basic



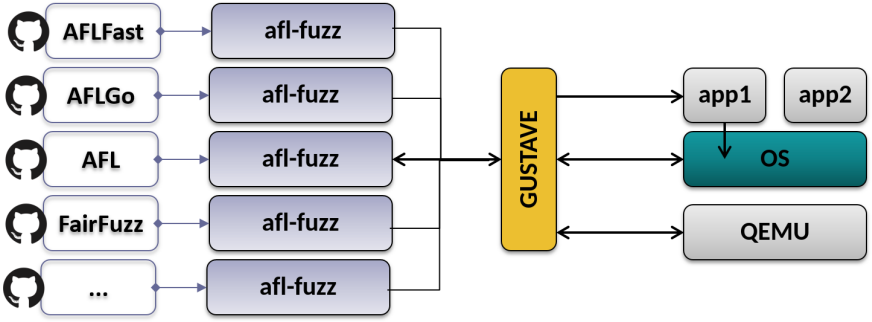
various architectures



multi-core


AFL
afl-fuzz -M
GUSTAVE
app1
app2
OS
QEMU
AFL
afl-fuzz -S
GUSTAVE
app1
app2
OS
QEMU

afl-fuzz forks



- build-time Instrumentation (`afl-gcc/afl-as`)
- initially design for various target configs:
  - basic apps and no scheduling
  - complex multi apps IPC scenario
  - focus a single system call

```
(target-src)$ 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%).
```

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

- source level instrumentation drawbacks
  - need specific firmware develop env
  - complex forge, Dev-lol-Ops
  - usually hard to obtain, implies a lot of contributors
- runtime binary instrumentation
  - hack into QEMU TCG
  - easier than estimated
  - open way to a lot of ideas/optimization/features

Gustave-NG

- QEMU Tiny Code Generator
- based on Paul Brook QOP code generator
- Guest insn → Intermediate Representation (IR) → Host insn
- Guest Basic Blocks → Translation Blocks (TB)
- powerful jit-compiler (block 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
```

- usually for system Guest insn
- helper functions developped in C
- builtin QEMU, generates host native `call`
- kind of `hypercall/paravirt_ops`

- Intel x86 64bits

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

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

- prevent `tb_find()` overhead
- QEMU/Guest ping pong for `next TB`
- fixing jump offsets inside TBs

- Intel x86 64bits

```
.7f5a0caf8190:  pushq    %r8
.7f5a0caf8192:  jmpq     *8(%rip)
...
.7f5a0caf81a0:  .quad  0x000055d62e3895a0
```

https://github.com/nccgroup/TriforceAFL

- OS fuzzer based on AFL/QEMU
  - enhance AFL qemu-mode
  - gen trace after each exec(TB)
  - inject Guest insn for AFL sync

```
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);
     }
}

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

- OS fuzzer based on AFL/QEMU
  - enhance AFL qemu-mode
  - gen trace after each exec(TB)
  - inject Guest insn for AFL sync
- Poorly designed
  - doesn't work with block chaining
  - impact AFL stability
  - impact performances
  - Linux centric: kmod, loader

```
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);
     }
}

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);
}
```

Better approach

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);
}
```

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
  - generate code coverage at IR level
  - during TBs generation
  - interesting but ...

```
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);
}
```

Even better approach

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);
}
```

```
/* during translation */
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);
}

/* during execution */
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;
}
```

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 (for AFLplusplus)
  - recent devs (07/2019)
  - use TCG helper for coverage
  - our implementation !

```
/* during translation */
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);
}

/* during execution */
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;
}
```

- 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,$0xfffffff8
movi_i32     tmp2,$0x0
brcond_i32   tmp1,tmp2,lt,$L0
movi_i32     tmp2,$0x14
add_i32      tmp1,r1,tmp2
```

- similar to AFL shim
- one index per TB
- update coverage trace bitmap
- same performances

- 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
```

Memory filtering

- How to monitor Guest kernel memory accesses ?
- Whatever the architecture or CPU execution mode ?
- With finest granularity (byte) ?
- Need to define legitimate accesses first ?

Detect kernel read/write out of “normal” areas !

- How to monitor Guest kernel memory accesses ?
- Whatever the architecture or CPU execution mode ?
- With finest granularity (byte) ?
- Need to define legitimate accesses first ?

Detect kernel read/write out of “normal” areas !

- TCG helpers for memory ops
- generic to the architecture
- finest control on memory

- PPC `stw r0, 4(r1)` → IR `qemu_st_xxx()`
- Lead to `tcg_out_qemu_st()` TCG backend-op
- QEMU memory model:
  - virtual TLBs for vCPU
  - guest physical to host physical addr
  - inlined in host generated code
  - slow and fast path concept
- Fast path: check for TLB hit and get host addr
- Slow path: use LDST labels to call a TCG helper: `tcg_out_qemu_st_slow_path()`

```
static void tcg_out_qemu_st(TCGContext *s, const TCGArg *args,
                            bool is64)
{
...

  /* try to find a filled TLB entry */
  tcg_out_tlb_load(s, addrlo, addrhi, mem_index, opc,
                   label_ptr, offsetof(CPUTLBEntry, addr_write));

  /* TLB Hit. So generate a physical guest memory access */
  tcg_out_qemu_st_direct(s, datalo, datahi, TCG_REG_L1, -1, 0, 0, opc);

  /* TLB Miss. Filled during tlb_load and redirect to soft-MMU */
  add_qemu_ldst_label(s, false, is64, oi, datalo, datahi, addrlo, addrhi,
                      s->code_ptr, label_ptr);
}
```

```
tcg_out_tlb_load:
.7ffff41888e9:   mov    %esp,%edi
.7ffff41888eb:   shr    $0x7,%edi
.7ffff41888ee:   and    0x338(%rbp),%edi
.7ffff41888f4:   add    0x388(%rbp),%rdi
.7ffff41888fb:   lea    0x3(%r12),%esi
.7ffff4188900:   and    $0xfffff000,%esi
.7ffff4188906:   cmp    0x4(%rdi),%esi
.7ffff4188909:   mov    %r12d,%esi
.7ffff418890c:   jne    0x7ffff418897f ;LDST
.7ffff4188912:   add    0x10(%rdi),%rsi

tcg_out_qemu_st_direct:
.7ffff4188925:   movbe  %ebx,(%rsi)
```

What has been done in GUSTAVE ?

- insert a call to a filtering helper
- into every QEMU mem access primitives
- dynamically check for legitimacy

What has been done in GUSTAVE ?

- insert a call to a filtering helper
- into every QEMU mem access primitives
- dynamically check for legitimacy

- use $O(1)$ bitmap for memory map
- 1bit/byte $\rightarrow$ 512MB host buffer for 4GB guest RAM
- learning phase to init the bitmap
- triggers `GUEST_PANICK` state for AFL classification

```
static void tcg_out_qemu_st(TCGContext *s, const TCGArg *args,
                            bool is64)
{
...
  tcg_out_tlb_load(s, addrlo, addrhi, mem_index, opc,
                   label_ptr, offsetof(CPUTLBEntry, addr_write));

  /* TLB Hit. So generate a physical guest memory access */
  tcg_out_qemu_st_filter(s, oi, datalo, datahi, addrlo, addrhi);
  tcg_out_qemu_st_direct(s, datalo, datahi, TCG_REG_L1, -1, 0, 0, opc);
}

static void tcg_out_qemu_st_filter(TCGContext *s, TCGMemOpIdx oi,
                                   TCGReg datalo, TCGReg datahi,
                                   TCGReg addrlo, TCGReg addrhi)
{
...
  tcg_out_call(s, qemu_st_filter_helpers[opc & (MO_BSWAP | MO_SIZE)]);
...
}
```

```
static void * const qemu_st_filter_helpers[16] = {
    [MO_UB]   = helper_filter_ret_stb_mmu,
    [MO_LEUW] = helper_filter_le_stw_mmu,
    [MO_LEUL] = helper_filter_le_stl_mmu,
    [MO_LEQ]  = helper_filter_le_stq_mmu,
    [MO_BEUW] = helper_filter_be_stw_mmu,
    [MO_BEUL] = helper_filter_be_stl_mmu,
    [MO_BEQ]  = helper_filter_be_stq_mmu,
};

void afl_tcg_filter_st(const char *from, CPUTLBEntry *entry,
                       uintptr_t haddr, target_ulong gaddr,
                       uint64_t val, int flags, int size)
{
...
    if (redzone_access(gaddr, size)) {
        qemu_log("%s: @ 0x"TARGET_FMT_lx
                 " H 0x%016"PRIx64
                 " G 0x"TARGET_FMT_lx
                 " = 0x%016"PRIx64
                 " flags:%d sz:%d\n",
                 from, afl_get_pc(&__global_afl->arch),
                 haddr, gaddr, val, flags, size);
        vm_stop(RUN_STATE_GUEST_PANICKED);
        cpu_loop_exit(CPU(__global_afl->arch.cpu));
    }
```

- tools to create redzone bitmap
- tmux scripts for multi-core AFL
- automatic environment setup
- replay crash cases !

```
$ pwd
crash_arinc/git/run/target

$ ls -l
-rw-rw-r-- 1 stf stf  617 Apr  3 00:25 Makefile
-rw------- 1 stf stf 3.3K Apr  2 23:58 README
-rw-rw-r-- 1 stf stf 1.3K Apr  3 17:16 TODO
lrwxrwxrwx 1 stf stf   22 Apr  2 21:23 afl-git -> ....
drwxrwxr-x 2 stf stf 4.0K Apr  2 22:08 binaries/
drwxrwxr-x 2 stf stf 4.0K Apr  2 17:46 config/
drwxrwxr-x 2 stf stf 4.0K Apr  2 23:43 gdb/
drwxrwxr-x 2 stf stf 4.0K Apr  2 14:46 input/
drwxrwxr-x 2 stf stf 4.0K Apr  2 23:26 mem_ranges/
drwxrwxr-x 3 stf stf 4.0K Apr  2 14:48 output/
lrwxrwxrwx 1 stf stf   17 Apr  2 21:23 qemu-git -> ....
drwxrwxr-x 3 stf stf 4.0K Jun 16 11:21 scripts/
```

```
$ ls -l mem_ranges/
-rw-rw-r-- 1 stf stf       149 Mar 13  2020 Makefile
-rw-rw-r-- 1 stf stf      2773 Apr  2 23:25 README
lrwxrwxrwx 1 stf stf        22 Apr  2 10:41 active.bitmap -> trace.bitmap
-rwxrwxr-x 1 stf stf     17592 Apr  2 10:22 bitmap
-rw-rw-r-- 1 stf stf      5031 Apr  2 10:22 bitmap.c
-rw-rw-r-- 1 stf stf       168 Apr  1 10:05 manual.mmap.sorted.bitmap
-rwxrwxr-x 1 stf stf      3419 Apr  1 17:37 mmranges.py
-rw-rw-r-- 1 stf stf   4169752 Apr  1 17:40 trace.bitmap
-rwxrwxr-x 1 stf stf     16752 Mar 16  2020 unlink_bitmap

$ ls -l scripts/
total 36
-rwxrwxr-x 1 stf stf  293 Dec 11 23:05 afl.sh
-rwx--x--x 1 stf stf  625 Dec 11 18:25 afl_tmux.sh
-rw------- 1 stf stf  825 Dec 10 15:08 check
-rwxrwxr-x 1 stf stf  278 Dec 11 18:25 cleanup.sh
-rwxrwxr-x 1 stf stf  126 Dec 11 18:36 config.sh
-rwxrwxr-x 1 stf stf 3311 Jan 15 14:14 env.sh
-rwxrwxr-x 1 stf stf  140 Dec 11 21:05 gdb.sh
-rwxrwxr-x 1 stf stf  523 Jan 11 12:21 replay.sh
-rwxrwxr-x 1 stf stf  116 Jan  5 15:07 run.sh
```

Results

- discovered vulnerabilities into proprietary OS
  - time and space partitioning properties
  - safety critical
- filter bitmap trapped illegal mem access done by kernel
- exploitability analysis thanks to replay-mode

```
> make replay
> FILE=$(find crashes_dir | shuf | head -n 1)
> ./scripts/replay.sh $FILE
```

```
afl-ppc: -- START TEST CASE --
\x06\x01\x04\xff\xff\xff\xff\x20
\x00\x00\x00\x00\x00\x00\x1a\x00
\x00\x01\x00\x00\x00\x0c\xeb\x80
afl-ppc: -- END TEST CASE --
afl-ppc: syscall [6|1|4]
afl-ppc: -- START PART STACK ARGS --
\xff\xff\xff\xff\x20\x00\x00\x00
\x00\x00\x00\x1a\x00\x00\x01\x00
\x00\x00\x0c\xeb\x80
afl-ppc: -- END PART STACK ARGS --
afl-ppc: vm exec end @ 0x005c01f2 (4)
afl-ppc: <-- resume vm (new test case)
afl-ppc: vm state running
io_writex: @ 0x0054d428 flags:3 sz:4
           H 0x0000000020000004
           G 0x0000000020000004
           = 0x0000000000000000
afl-ppc: vm state guest-panicked
afl-ppc: --> vm abort() !
```

Conclusion

Short-term

- deep dive into TCG raised lot of ideas
- GUSTAVE behavior as a QEMU fork, not a specific board
- goal is to support all available architectures
- still need Syscall ABI translation

Optimizations

- keep TCG cache for kernel code
- snapshot subsystem (restore minimal)
- use host CPU features (ie. Intel PT)

A few questions arose

- think about code coverage ratio (very low) ?
- what could be done smarter than fuzzing ?
- how to be less restrictive on memory interception ?
- setup ideal/attack prone target firmware environment
- Gustave is flexible, can interfact with other tools
  - redqueen, input-to-state correspondence
    `https://github.com/RUB-SysSec/redqueen`
  - syzkaller, type inference
    `https://github.com/google/syzkaller`
  - honggfuzz, trendy
    `https://google.github.io/honggfuzz`