注：本文取自腾讯科恩高校合作Fuzzing方向讨论群 - 2021年3月15日

大致读了一下论文（CCS 17 - Designing New Operating Primitives to Improve Fuzzing Performance），这个 snapshot 新调用看起来主要是为了提升 fork 的多核扩展性，加上这个模块之前停止更新了，我们机器的内核版本又比较新，就没有尝试。不过我们似乎找到一个不错的优化方法，在这里给大家分享一下。

-- [介绍

*简单*概括一下 QEMU 内置的虚拟化设备 fuzzer，它是基于 QEMU 内置的设备测试框架 QTest 实现的，其工作原理是从 fuzzer 接受变异数据块，根据提前定义的标志符将其分为不同的 IO操作码+操作数，通过 QTest 输入操作码+操作数对设备进行 fuzz。操作码有以下十种：

1. IO port in
2. IO port out
3. MMIO write
4. MMIO read
5. PCI configure space write
6. PCI configure space read
7. disable PCI configure
8. clock step
9. add DMA pattern
10. clear DMA pattern

这里解释一下 9 操作，他们主要是为了对 DMA 操作的目的地址进行动态的进行 fuzz —— fuzzer 会提前 hook 内存中的 DMA 操作，当 MMIO 操作激发 DMA 后，fuzzer 会实时的根据 DMA pattern判断该操作是否合法，然后在目的内存块填入畸形的数据，继续进行 DMA。

-- [现象

通过一些监控手段（Linux perf），我们发现 fuzzer 的大部分 CPU 周期都放在了 fork 相关的操作上了（kernel: copy_pte_range, exit_mmap），占用的比例近到80%：

```
Samples: 1M of event 'cpu-cycles', Event count (approx.): 1451076260479
  Children      Self  Command          Shared Object            Symbol
-   78.44%     0.06%  qemu-fuzz-i386   [kernel.kallsyms]        [k]
entry_SYSCALL_64_after_hwframe
   - 78.38% entry_SYSCALL_64_after_hwframe
      - 78.31% do_syscall_64
         - 38.13% __syscall_return_slowpath
            - 38.12% __prepare_exit_to_usermode
               - 38.09% exit_to_usermode_loop
                  - 38.09% do_signal
                     - 38.09% get_signal
                        - 38.09% do_group_exit
                           - 38.09% do_exit
                              + 37.84% exit_mm
         - 30.69% __x64_sys_clone
            - 30.69% __do_sys_clone
               - 30.68% _do_fork
                  - 30.54% copy_process
                     - 29.78% dup_mm
                        - 29.70% dup_mmap
                           + 27.11% copy_page_range
```

所以为了加速，我们可以选择加快 fork 的速度（上面这篇论文），也可以让每个 fork 的效果更好，或者减少不必要的 fork ，这也是我们采用的方法。

我们记录下测试4小时后主要操作符生成的比例，结果如下（测试设备为 QEMU 默认的网卡 e1000）：

各操作次数
Forks 1510353
Total:Real 32941266:21224748 64.43%（有效操作占比）
in 2465725 11.62%（该操作符在所有有效操作中的占比）
out 466618 2.20%
read 978025 4.61%
write 1069104 5.04%
pci_read 268340 1.26%
pci_write 1873546 8.83%
clock_step 2228757 10.50%
disable_pci 4656019 21.94%
add_dma 245832 1.16%
clear_dma 6974054 32.86%

覆盖度情况（libfuzzer 的输出，https://llvm.org/docs/LibFuzzer.html#output）
#1511404 cov: 3931 ft: 2729

其中，Total:Real代表了在所有生成的操作符中，真正进入 fuzzing 的操作符比例，这是因为不同操作符的操作数长度不一样，但 fuzzer 在对数据插入标志符的时候是任意的，所以有的操作符对应的操作数不满足要求，需要被丢弃。

-- [分析

从上面的数据我们可以得出两个结论：

一是我们生成有效操作符的效率并不高，这会影响程序分析和执行的速度，也会影响 fuzzer 变异的效率（即使libfuzzer采用了和 AFL 一样的 effector map，也无法完全避免对无效操作数的变异尝试）。

二是大部分的操作符非直接IO的操作，例如 clear_dma 和 disable_pci。这也是不合理的 —— 我们统计了 QEMU 的 bug-tracker 上 OSS-FUZZ 的 52 份报告，其中能产生 crash 的主要操作符比例如下：

MMIO write 1020 65.1%
PCI configure write 259 16.5%
clock step 165 10.5%
IO port out 33 2.1%

可见，如果从依从经验的角度出发，我们应该偏重 IO 输出操作和时钟步进操作。

-- [解决

从减少 fork 次数，增加有效操作符，提升 IO 输出操作的目的出发，我们对不同操作码赋予不同的权值，并把对有效操作码的判定提前到了 fork 之前，将无效操作码其也看做为一类操作码。对于无效操作码和现在 fuzzer 中占比较高的操作，我们均予以负值，对于 MMIO write 这类需要提升比例的操作我们给与正值。只有当当前输入总的权值为正的时候，我们才进行 fork。其中核心逻辑如下:

```
/* -- Wights for each ops to guide our forks -- */
#define FUZZ_BAD_OP_PENALTY (-4)
#define FUZZ_OP_IN_SCORE (rand() % 2 ? 0 : -1) /* -0.5 */
#define FUZZ_OP_OUT_SCORE (0)
#define FUZZ_OP_READ_SCORE (rand() % 4 ? 0 : -1) /* -0.25 */
#define FUZZ_OP_WRITE_SCORE (1)
#define FUZZ_OP_PCI_READ_SCORE (0)
#define FUZZ_OP_PCI_WRITE_SCORE (rand() % 4 ? 0 : -1) /* -0.25 */
#define FUZZ_OP_DISABLE_PCI_SCORE (-1)
#define FUZZ_OP_ADD_DMA_PATTERN_SCORE (0)
#define FUZZ_OP_CLEAR_DMA_PATTERNS_SCORE (-2)
```

```
#define FUZZ_OP_CLOCK_STEP_SCORE (0)
```

注意到虽然我们要提高 PCI configure write 的比例，我们还是给它-0.25的负值，这是因为我们压制了 disable PCI ，这会导致 PCI 对应操作尤其是写比例大幅度提升，同时，在大部分设备 fuzzing 初期，PCI 写操作对覆盖度的影响都会比较大（fuzzer会保留这些操作），为了避免“早熟”，我们也给与一定的压制。

```
/* -- calculate the weight of current input -- */
    long score = 0;
    while (cmd && Size) {
        /* Get the length until the next command or end of input */
        nextcmd = memmem(cmd, Size, SEPARATOR, strlen(SEPARATOR));
        cmd_len = nextcmd ? nextcmd - cmd : Size;
        if (cmd_len > 0) {
            /* Interpret the first byte of the command as an opcode */
            op = *cmd % (sizeof(ops) / sizeof((ops)[0]));
            score += ops[op](s, cmd + 1, cmd_len - 1, true);
        }
        /* Advance to the next command */
        cmd = nextcmd ? nextcmd + sizeof(SEPARATOR) - 1 : nextcmd;
        Size = Size - (cmd_len + sizeof(SEPARATOR) - 1);
    }
    if (score < 0) return;

/* -- Port IO write -- */
static int op_write(QTestState *s, const unsigned char * data, size_t len, bool check)
{
    enum Sizes {Byte, Word, Long, Quad, end_sizes};
    struct {
        uint8_t size;
        uint8_t base;
        uint32_t offset;
        uint64_t value;
    } a;
    address_range abs;

    if (len < sizeof(a))
        return FUZZ_BAD_OP_PENALTY;
    memcpy(&a, data, sizeof(a));

    if (get_mmio_address(&abs, a.base, a.offset) == 0)
        return FUZZ_BAD_OP_PENALTY;

    if (check)
        return FUZZ_OP_WRITE_SCORE;
```

-- [评估

同样是对于 e1000 网卡，我们再次进行 fuzzing，结果12分钟后（20×加速比）我们就达到了之前4小时的块覆盖度：

```
各操作次数
Forks           128722
Total:Real      340323:313735   92.19%
in              19479           6.21%
out             22629           7.21%
read            16298           5.19%
write           60858           19.40%
pci_read        31621           10.08%
pci_write       93569           29.82%
clock_step      52906           16.86%
disable_pci     6272            2.00%
add_dma         6616            2.11%
clear_dma       3490            1.11%
```

覆盖度情况
#330313 cov: 3947 ft: 23309

可以看到，我们的有效操作码的比例，以及IO输出操作的比例都有了很大提高 :)

-- [后记

各个操作的配比跟设备行为密切相关，这里只是根据现有经验做了一下手动调整，如果能够自动化的调整会更好。

归根到底，对虚拟化设备IO的 fuzzing 是一种 stateful API fuzzing，我们这里如果采用结构化 fuzzing（例如 protobuf：oneof commands）和动态调度算法（例如 USENIX sec 19, MOPT），效果肯定会更好，但谁有时间呢？
Happy Fuzzing!

Thank you.
  Qiuhao Li