不只二进制翻译

探究 Box64 实现细节

刘阳
PLCT 实验室
2024 年 07 月

关于我

- ksco (Yang Liu) · GitHub
- RISC-V 和 LoongArch JIT 后端主要贡献者之一
- AOSC OS 用户

没听说过 Box64?

- x86-64 用户态模拟器
  - QEMU user、Rosetta 2、Fex EMU、LATX
- https://github.com/ptitSeb/box64
- 最初的代码拷贝修改自 Box86
  - 同作者的 x86 模拟器，仅支持 armhf
- 动态重编译器（DynaRec）：AArch64、RISC-V、LoongArch
- libwrapping：200+ 常用的库

差异化

	box64	QEMU user	Rosetta 2	Fex EMU	LATX
性能	不错	能用	最好	还行	不错
开源	✅	✅	⛔	✅	⛔
libwrap	一等公民	不支持	N/A	有限支持	有限支持
JIT 支持架构	AArch64 RISC-V LoongArch	所有架构	AArch64	AArch64	LoongArch
AOT	⛔	⛔	✅	⛔	有限支持

各个模块的简单介绍

libwrapping

- Linux Userspace x86_64 Emulator with a twist
- 自己就是“动态链接器”，不依赖 ld-linux-x86-64.so.2
- Emulated libs 和 native libs
- 将对动态链接库的函数调用转发到本地的动态链接库中

解释器

- 若干个巨大的嵌套的 switch 语句

```
// 66 0F 38 0C [PBLENDPS Gx, Ex, Ib]
#define F8 *(uint8_t*)(addr++)
case 0x0C:
    nextop = F8;
    GETEX(1);
    GETGX;
    tmp8u = F8;
    for (int i = 0; i < 4; ++i) {
        if(tmp8u & (1 << i)) GX->ud[i] = EX->ud[i];
    }
    break;
```

- 支持的指令最全，作为兜底
- 负责运行 `__libc_start_main` 之前的代码
- 用于处理 TF（trap flag）、SMC 等

DynaRec

- 同样是若干个巨大的嵌套 switch 语句
- 支持 ARM64、RV64 以及 LA64
- 朴素的多趟代码生成，无 IR，无优化器
- 朴素的寄存器分配
  - 每个 x86 整形寄存器固定的映射到一个 native GPR，剩余 GPR 作为 scratch 寄存器
  - 有一个简单的分配器用于映射浮点和 SIMD / Vector 寄存器
- eflags 依赖计算
- 等级可调节但强内存模型模拟
- 多样化的调试手段

```
13322|Emitting 228 bytes for 86 x64 bytes ( => /home/ksco/GOGGames/SOMA/game/Soma.bin.x86_64:_ZN3hpl4cGui12SendKeyPressERKNS_9cKeyPressE)
0xc9e770: 48 83 EC 58 sub rsp, 0x58
0x7fff6351de38: 2 emitted opcodes, inst=0, barrier=0 state=3/0(0), set=3F/0, use=0, need=0/0, sm=0/0, last_ip=0xc9e770
        03816007        ORI             x3_r7, xZR, 0x58
        00119e10        SUB.D           xRSP_r16, xRSP_r16, x3_r7
0xc9e774: 48 8B 7F 40 mov rdi, [rdi+0x40]
0x7fff6351de40: 1 emitted opcodes, inst=1, barrier=0 state=0/1(0), set=0/0, use=0, need=0/0, sm=0/0, pred=0, last_ip=0xc9e770
        28c10273        LD.D            xRDI_r19, xRDI_r19, 64
0xc9e778: 48 85 FF test rdi, rdi
0x7fff6351de44: 2 emitted opcodes, inst=2, barrier=0 state=3/1(0), set=3F/8, use=0, need=0/8, sm=0/0, pred=1, last_ip=0xc9e770
        29913080        ST.W            xZR, xEmu, 1100
        003fce73        X64AND.D        xRDI_r19, xRDI_r19
0xc9e77b: 74 42 jz 0x0000000000C9E7BF
0x7fff6351de4c: 3 emitted opcodes, inst=3, barrier=0 state=0/1(1), set=0/0, use=8, need=8/0, sm=0/0, pred=2, jmp=17, last_ip=0xc9e770
        00369005        X64SETJ         x1_r5, 0x4
        440078a0        BNEZ            x1_r5, 120
        03400000        NOP
0xc9e77d: C6 44 24 08 01 mov byte ptr [rsp+0x08], 0x01
0x7fff6351de58: 2 emitted opcodes, inst=4, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/0, pred=3, last_ip=0xc9e770
        02c00407        ADDI.D          x3_r7, xZR, 1
        29002207        ST.B            x3_r7, xRSP_r16, 8
0xc9e782: C6 44 24 09 00 mov byte ptr [rsp+0x09], 0x00
0x7fff6351de60: 1 emitted opcodes, inst=5, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=4, last_ip=0xc9e770
        29002600        ST.B            xZR, xRSP_r16, 9
0xc9e787: C7 44 24 0C 01 00 00 00 mov dword ptr [rsp+0x0C], 0x01
0x7fff6351de64: 3 emitted opcodes, inst=6, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=5, last_ip=0xc9e770
        02c03206        ADDI.D          x2_r6, xRSP_r16, 12
        03800407        ORI             x3_r7, xZR, 0x1
        298000c7        ST.W            x3_r7, x2_r6, 0
0xc9e78f: 48 C7 04 24 50 46 9C 01 mov qword ptr [rsp], 0x19C4650
0x7fff6351de70: 3 emitted opcodes, inst=7, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=6, last_ip=0xc9e770
        14033887        LU12I.W         x3_r7, 0x19c4
        039940e7        ORI             x3_r7, x3_r7, 0x650
        29c00207        ST.D            x3_r7, xRSP_r16, 0
0xc9e797: 48 C7 44 24 38 00 00 00 00 mov qword ptr [rsp+0x38], 0x00
0x7fff6351de7c: 2 emitted opcodes, inst=8, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=7, last_ip=0xc9e770
        02c0e206        ADDI.D          x2_r6, xRSP_r16, 56
        29c000c0        ST.D            xZR, x2_r6, 0
0xc9e7a0: 8B 46 08 mov eax, [rsi+0x08]
0x7fff6351de84: 1 emitted opcodes, inst=9, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=8, last_ip=0xc9e770
        2a80224c        LD.WU           xRAX_r12, xRSI_r18, 8
0xc9e7a3: 89 44 24 2C mov [rsp+0x2C], eax
0x7fff6351de88: 1 emitted opcodes, inst=10, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=9, last_ip=0xc9e770
        2980b20c        ST.W            xRAX_r12, xRSP_r16, 44
0xc9e7a7: 48 8B 06 mov rax, [rsi]
0x7fff6351de8c: 1 emitted opcodes, inst=11, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=10, last_ip=0xc9e770
        28c0024c        LD.D            xRAX_r12, xRSI_r18, 0
0xc9e7aa: 48 89 44 24 24 mov [rsp+0x24], rax
0x7fff6351de90: 1 emitted opcodes, inst=12, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=11, last_ip=0xc9e770
        29c0920c        ST.D            xRAX_r12, xRSP_r16, 36
0xc9e7af: 48 8D 14 24 lea rdx, [rsp]
0x7fff6351de94: 1 emitted opcodes, inst=13, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=12, last_ip=0xc9e770
        02c0020e        ADDI.D          xRDX_r14, xRSP_r16, 0
0xc9e7b3: BE 11 00 00 00 mov esi, 0x11
0x7fff6351de98: 1 emitted opcodes, inst=14, barrier=0 state=0/1(1), set=0/0, use=0, need=0/0, sm=0/1, pred=13, last_ip=0xc9e770
        03804412        ORI             xRSI_r18, xZR, 0x11
```

几个有趣的实现细节

#1

如何将函数调用转发到 Native 库?

原生执行的 x86_64 程序是如何调用外部函数的?

```
call   1030 <puts@plt>
```

PLT

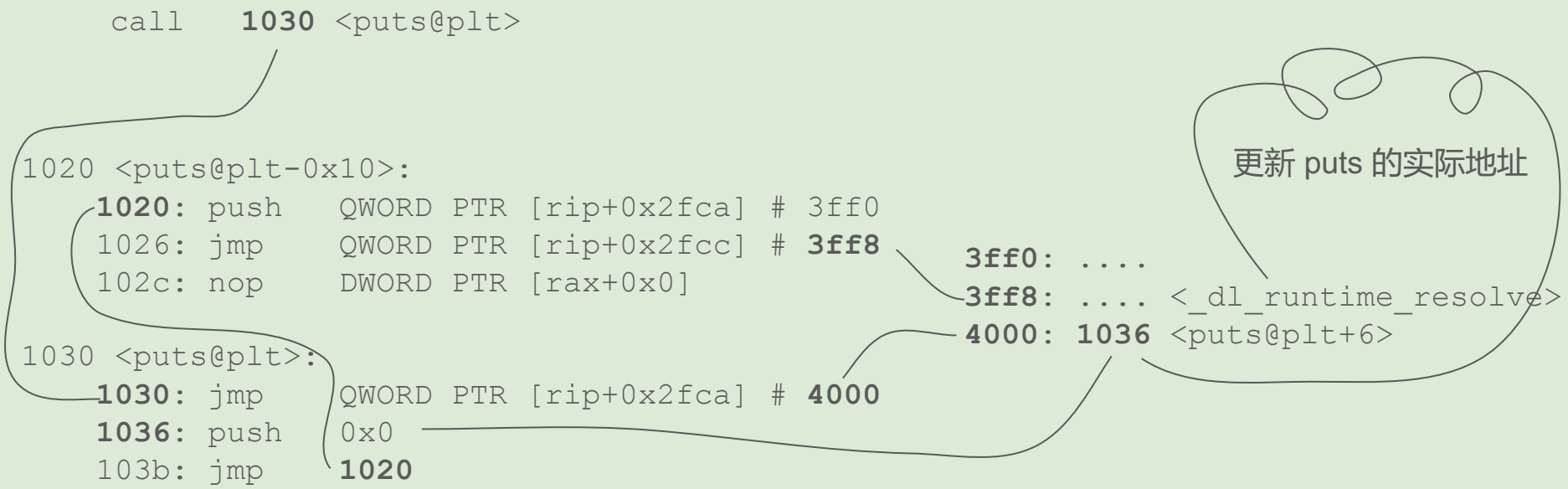
```
1020 <puts@plt-0x10>:
    1020: push   QWORD PTR [rip+0x2fca] # 3ff0
    1026: jmp    QWORD PTR [rip+0x2fcc] # 3ff8
    102c: nop    DWORD PTR [rax+0x0]

1030 <puts@plt>:
    1030: jmp    QWORD PTR [rip+0x2fca] # 4000
    1036: push   0x0
    103b: jmp    1020
```

GOT

```
3ff0: ....
3ff8: .... <_dl_runtime_resolve>
4000: 1036 <puts@plt+6>
```

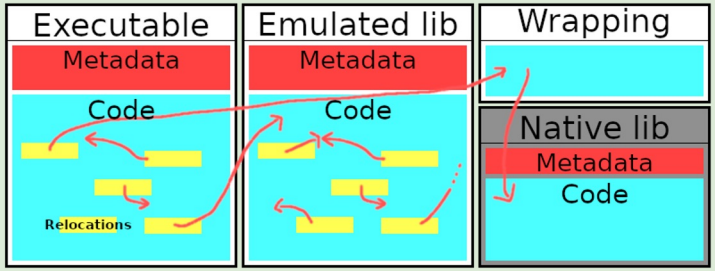
原生执行的 x86_64 程序是如何调用外部函数的?



"All problems in computer science can be solved by another level of indirection"

~~如果不行，那就再加一层。~~

Box64 对这个过程的魔改




```
    call   1030 <puts@plt>

1020 <puts@plt-0x10>:
   1020: push   QWORD PTR [rip+0x2fca] # 3ff0
   1026: jmp    QWORD PTR [rip+0x2fcc] # 3ff8
   102c: nop    DWORD PTR [rax+0x0]

1030 <puts@plt>:
   1030: jmp    QWORD PTR [rip+0x2fca] # 4000
   1036: push   0x0
   103b: jmp    1020

3ff0: ....
3ff8: .... <PltResolver>
4000: 1036 <puts@plt+6>
```

更新为 bridge 的地址

“All problems in computer science can be solved by another level of indirection”

~~如果不行，那就再加一层。~~

Bridge?

```
typedef void (*wrapper_t)(x64emu_t* emu, uintptr_t fnc);

struct {
  uint8_t CC;      // INT3
  uint8_t S, C;    // 'S' 'C', just a signature
  wrapper_t w;     // wrapper
  uintptr_t f;     // the function for the wrapper
}

typedef void (*vFii_t)(int32_t, int32_t);

void vFii(x64emu_t *emu, uintptr_t fcn) {
    vFii_t fn = (vFii_t)fcn;
    fn((int32_t)R_RDI, (int32_t)R_RSI);
}
```

#2

那如果 Native 函数有 Callback 呢?

那就再加一层 Redirection

Wrapper 函数不再直接调用 native 函数，而是调用 my_ 函数

```
EXPORT void my3_CRYPTO_set_id_callback(x64emu_t* emu, void* cb)
{
    my->CRYPTO_set_id_callback(find_id_func_Fct(cb));
}
```

那就再加一层 Redirection

```
#define SUPER() GO(0) GO(1) GO(2) GO(3) GO(4)
#define GO(A)                                                          \
    static uintptr_t my3_id_func_fct_##A = 0;                          \
    static unsigned long my3_id_func_##A()                             \
    {                                                                  \
        return (unsigned long)RunFunctionFmt(my3_id_func_fct_##A, ""); \
    }
SUPER()
#undef GO
static void* find_id_func_Fct(void* fct)
{
    if(!fct) return NULL;
    void* p;
    if((p = GetNativeFnc((uintptr_t)fct))) return p;
    #define GO(A) if(my3_id_func_fct_##A == (uintptr_t)fct) return my3_id_func_##A;
    SUPER()
    #undef GO
    #define GO(A)                                  \
    if (my3_id_func_fct_##A == 0) {                \
        my3_id_func_fct_##A = (uintptr_t)fct; \
        return my3_id_func_##A;                    \
    }
    SUPER()
    #undef GO
    printf_log(LOG_NONE, "Warning, no more slot for libcrypto id_func callback\n");
    return NULL;
}
```

#3

多趟代码生成

多趟代码生成

```
case 0x01:
    INST_NAME("ADD Ed, Gd");
    SETFLAGS(X_ALL, SF_SET_PENDING);
    nextop = F8;
    GETGD;
    GETED(0);
    emit_add32(dyn, ninst, rex, ed, gd, x3, x4, x5);
    WBACK;
    break;
```

通过宏将同一套代码在四个 pass 重复利用。

0. 计算 block 中 x86 指令数量

1. 标记跳转是否为块内跳转，计算 eflags 依赖关系*

2. 计算 native 指令数量

3. 实际的指令生成

* 每条指令需要标记依赖哪些 flags，设置哪些 flags

#4

CALL/RET 优化

CALL/RET 优化

假设：

CALL → 函数调用；RET → 函数返回

正常实现方式：

CALL → 查表（慢）→ 跳转 → … → RET → 查表（慢） → 返回

开启 CALL/RET 优化

CALL → 压栈 → 查表（慢）→ 跳转 → … → RET → 弹栈比对 → 返回

将 x64 地址和 native 地址一起压栈

比对 x64 地址和 RET 地址是否一致

#5

信号处理

以 SIGSEGV 为例

- 注册全局 signal handler 代理所有的 SIGSEGV
- 判断 PC 是否来自 JIT 区域
- 恢复出事现场寄存器状态
- 调用 guest signal handler

How?

- **寄存器一一映射**
- **同一条指令对于寄存器的写入一定在对内存的写入之后**

因此，任意时刻 x86 寄存器的状态在访存之前一定是精确的

#6

Self-Modifying Code™

Self-Modifying Code™

- guest 代码页设置为只读
- signal handler 中判断 si_addr 是否属于 guest 代码页
- 设置 dynablock 为 dirty
- 对应的 guest 代码页设置为可写
- `siglongjump` 返回到解释器，使用解释器执行 SMC 代码
- 解释器遇到跳转指令回到 DynaRec
- 标记为脏的 dynablock 在执行前会验证 CRC，并重建

#7

Co-simulator

Co-simulator

- 调试用基础设施
- 开启后，DynaRec 会在每一条指令的后面插入 **`test_step()`** 函数调用，该函数会执行该指令的解释器版本，然后对比两条指令执行后微架构状态是否一致

```
Warning, difference between x64 Interpreter and Dynarec in 0x10000c82d (66 0f ef c1 0f 29 85 40)

=====================================

DIFF: Dynarec |  Interpreter

----------------------

XMM[00]: 01100f0a06810708-00ff802015050100 | 33d00faf54f383f6-81fc8221157a81ff
```

Q&A