Initial message

Welcome to ./babyrev_level22.0!
This challenge is an custom emulator. It emulates a completely custom architecture that we call "Yan85"! You'll have to understand the emulator to understand the architecture, and you'll have to understand the architecture to understand the code being emulated, and you will have to understand that code to get the flag. Good luck!
This level is a full Yan85 emulator. You'll have to reason about yancode, and the implications of how the emulator interprets it! This challenge is special! It randomizes the Yan85 VM based on the value of the flag. This means that there is no way for you to know the opcode and argument encodings...
Keep in mind that the encoding that you observe in practice mode is going to be different than the actual encoding, because the practice mode flag is different. How will you adapt? Is there maybe a clever side channel you can utilize?

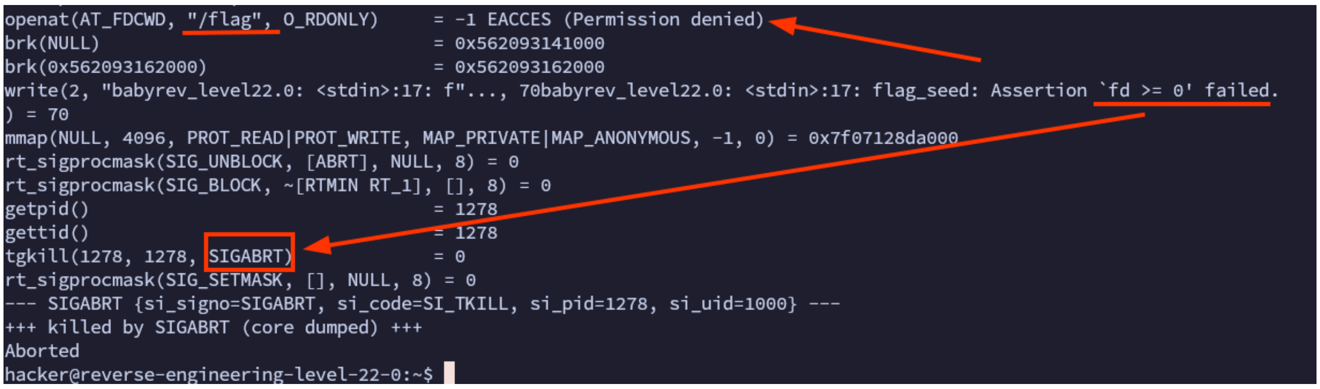
... Done! VM is randomized!

What to do

Understand emulator -> understand architecture -> understand the code being emulated -> understand that code to get the flag.

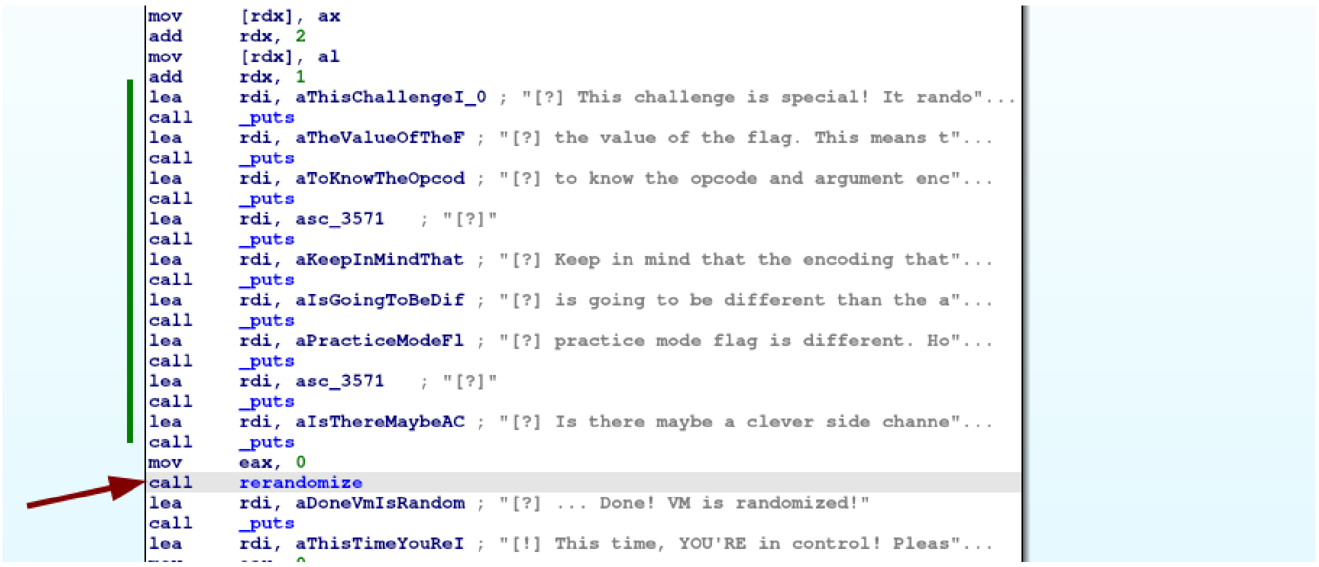
After digging up and having understood the most important structure, we need to figure out the key-dependent initial configuration on the remote machine.

Findings



```
openat(AT_FDCWD, "/flag", O_RDONLY)     = -1 EACCES (Permission denied)
brk(NULL)                               = 0x562093141000
brk(0x562093162000)                     = 0x562093162000
write(2, "babyrev_level22.0: <stdin>:17: f"..., 70babyrev_level22.0: <stdin>:17: flag_seed: Assertion `fd >= 0' failed.
) = 70
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f07128da000
rt_sigprocmask(SIG_UNBLOCK, [ABRT], NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, ~[RTMIN RT_1], [], 8) = 0
getpid()                                = 1278
gettid()                                = 1278
tgkill(1278, 1278, SIGABRT)             = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
--- SIGABRT {si_signo=SIGABRT, si_code=SI_TKILL, si_pid=1278, si_uid=1000} ---
+++ killed by SIGABRT (core dumped) +++
Aborted
hacker@reverse-engineering-level-22-0:~$
```

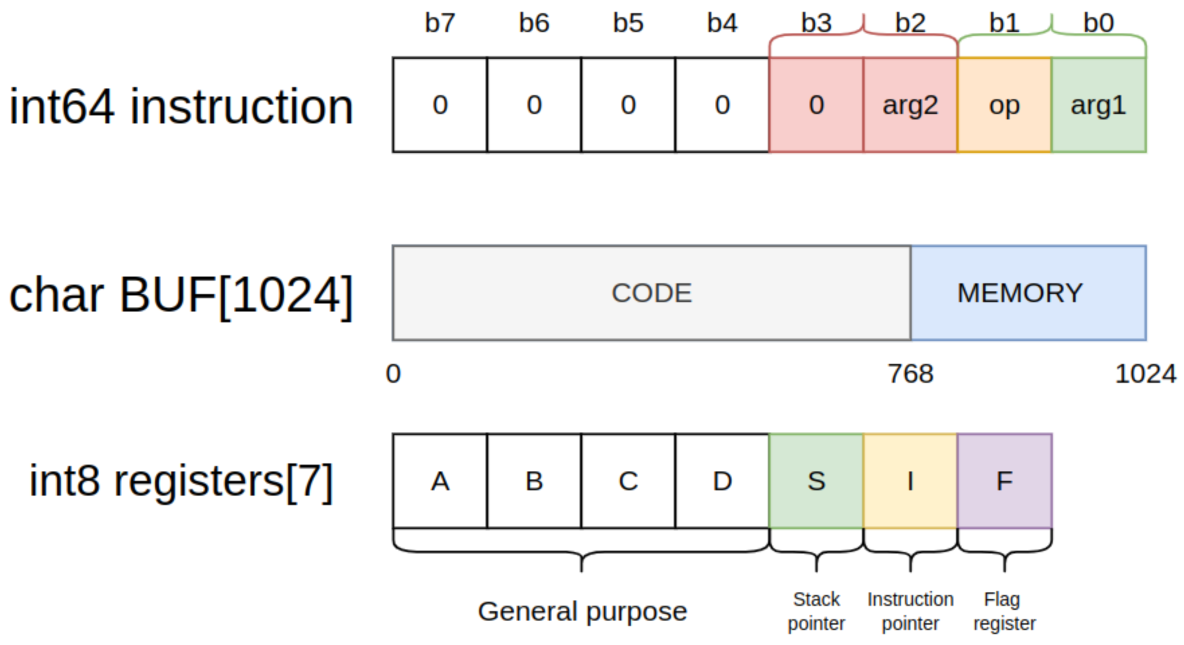
Using strace we see that the challenge requires the file '/flag' otherwise aborts.



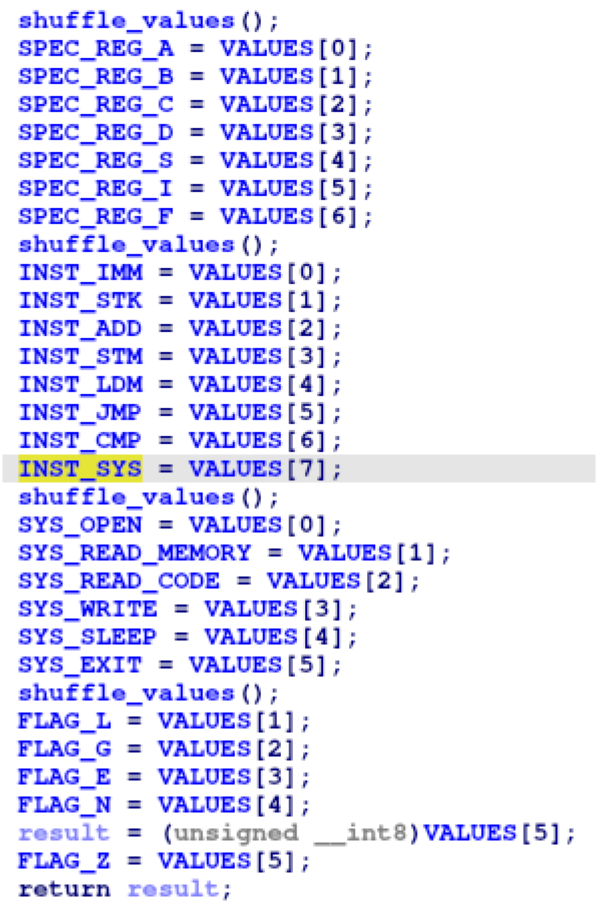
```
mov     [rdx], ax
add     rdx, 2
mov     [rdx], al
add     rdx, 1
lea     rdi, aThisChallengeI_0 ; "[?] This challenge is special! It rando"...
call    _puts
lea     rdi, aTheValueOfTheF ; "[?] the value of the flag. This means t"...
call    _puts
lea     rdi, aToKnowTheOpcod ; "[?] to know the opcode and argument enc"...
call    _puts
lea     rdi, asc_3571    ; "[?]"
call    _puts
lea     rdi, aKeepInMindThat ; "[?] Keep in mind that the encoding that"...
call    _puts
lea     rdi, aIsGoingToBeDif ; "[?] is going to be different than the a"...
call    _puts
lea     rdi, aPracticeModeFl ; "[?] practice mode flag is different. Ho"...
call    _puts
lea     rdi, asc_3571    ; "[?]"
call    _puts
lea     rdi, aIsThereMaybeAC ; "[?] Is there maybe a clever side channe"...
call    _puts
mov     eax, 0
call    rerandomize
lea     rdi, aDoneVmIsRandom ; "[?] ... Done! VM is randomized!"
call    _puts
lea     rdi, aThisTimeYouReI ; "[!] This time, YOU'RE in control! Pleas"...
```

After some initial prints, the code calls rerandomize.

Relevant structures



rerandomize()



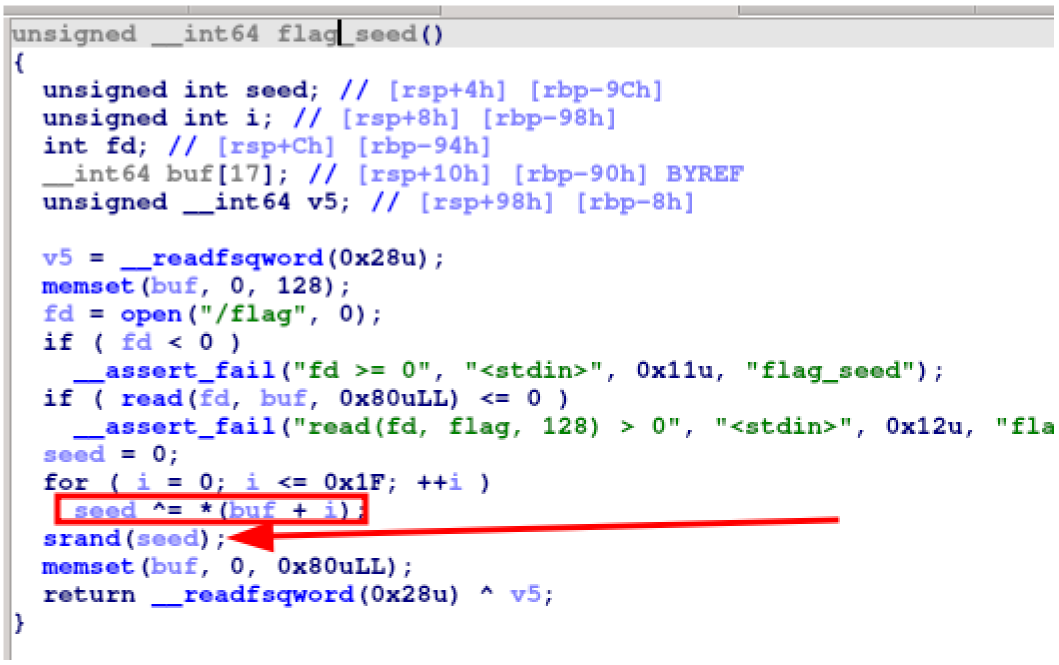
```
shuffle_values();
SPEC_REG_A = VALUES[0];
SPEC_REG_B = VALUES[1];
SPEC_REG_C = VALUES[2];
SPEC_REG_D = VALUES[3];
SPEC_REG_S = VALUES[4];
SPEC_REG_I = VALUES[5];
SPEC_REG_F = VALUES[6];
shuffle_values();
INST_IMM = VALUES[0];
INST_STK = VALUES[1];
INST_ADD = VALUES[2];
INST_STM = VALUES[3];
INST_LDM = VALUES[4];
INST_JMP = VALUES[5];
INST_CMP = VALUES[6];
INST_SYS = VALUES[7];
shuffle_values();
SYS_OPEN = VALUES[0];
SYS_READ_MEMORY = VALUES[1];
SYS_READ_CODE = VALUES[2];
SYS_WRITE = VALUES[3];
SYS_SLEEP = VALUES[4];
SYS_EXIT = VALUES[5];
shuffle_values();
FLAG_L = VALUES[1];
FLAG_G = VALUES[2];
FLAG_E = VALUES[3];
FLAG_N = VALUES[4];
result = (unsigned __int8)VALUES[5];
FLAG_Z = VALUES[5];
return result;
```

shuffles and assings values to:

- Registers
- instructions
- syscalls
- cpu flags

shuffle_values()

flag_seed()

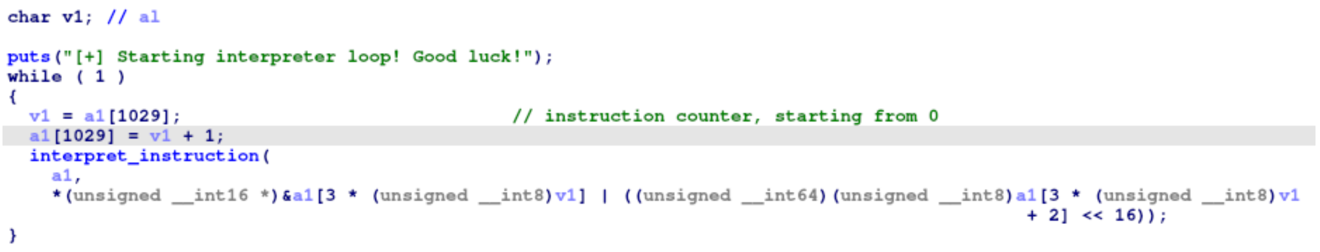


```
unsigned __int64 flag_seed()
{
  unsigned int seed; // [rsp+4h] [rbp-9Ch]
  unsigned int i; // [rsp+8h] [rbp-98h]
  int fd; // [rsp+Ch] [rbp-94h]
  __int64 buf[17]; // [rsp+10h] [rbp-90h] BYREF
  unsigned __int64 v5; // [rsp+98h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  memset(buf, 0, 128);
  fd = open("/flag", 0);
  if ( fd < 0 )
    __assert_fail("fd >= 0", "<stdin>", 0x11u, "flag_seed");
  if ( read(fd, buf, 0x80uLL) <= 0 )
    __assert_fail("read(fd, flag, 128) > 0", "<stdin>", 0x12u, "fla
  seed = 0;
  for ( i = 0; i <= 0x1F; ++i )
    seed ^= *(buf + i);
  srand(seed);
  memset(buf, 0, 0x80uLL);
  return __readfsqword(0x28u) ^ v5;
}
```

We can see that the seed is computed from the flag, so it is actually pseudo random with a fixed 'architecture version' for a given flag.

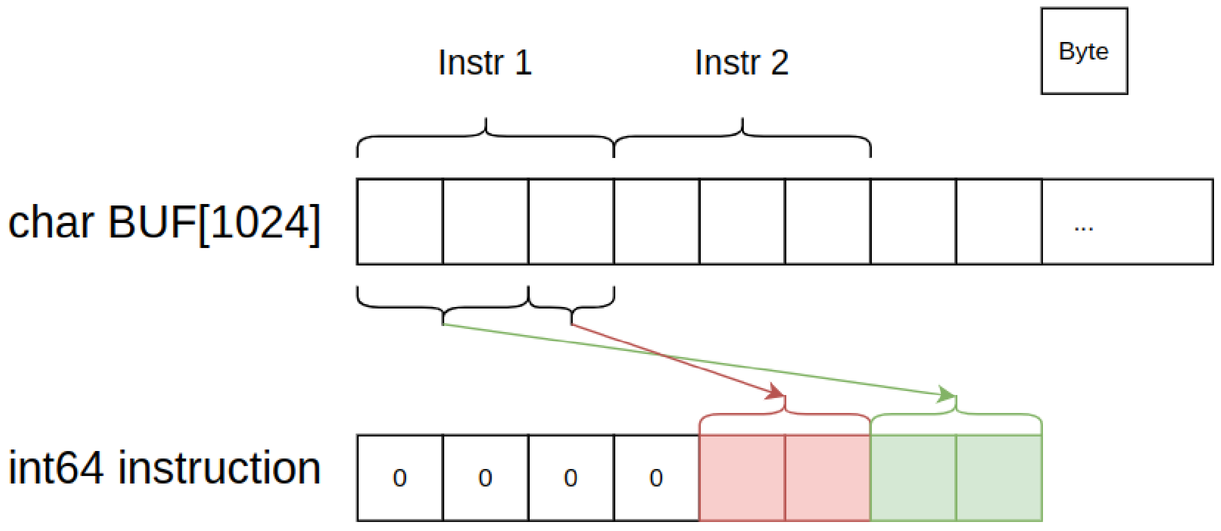
interpreter_loop(char* buf_ptr)



```
char v1; // a1

puts("[+] Starting interpreter loop! Good luck!");
while ( 1 )
{
  v1 = a1[1029];                                 // instruction counter, starting from 0
  a1[1029] = v1 + 1;
  interpret_instruction(
    a1,
    *(unsigned __int16 *)&a1[3 * (unsigned __int8)v1] | ((unsigned __int64)(unsigned __int8)a1[3 * (unsigned __int8)v1
                                                                                                 + 2] << 16));
}
```

The interpreter loop is an infinite loop that increments the instruction counter. Each time it parses three bytes from the buffer, and does the following:

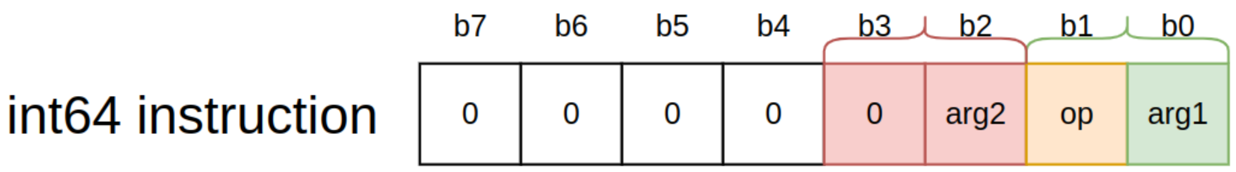


interpreter_instruction(char* buf_ptr, int64 instruction)

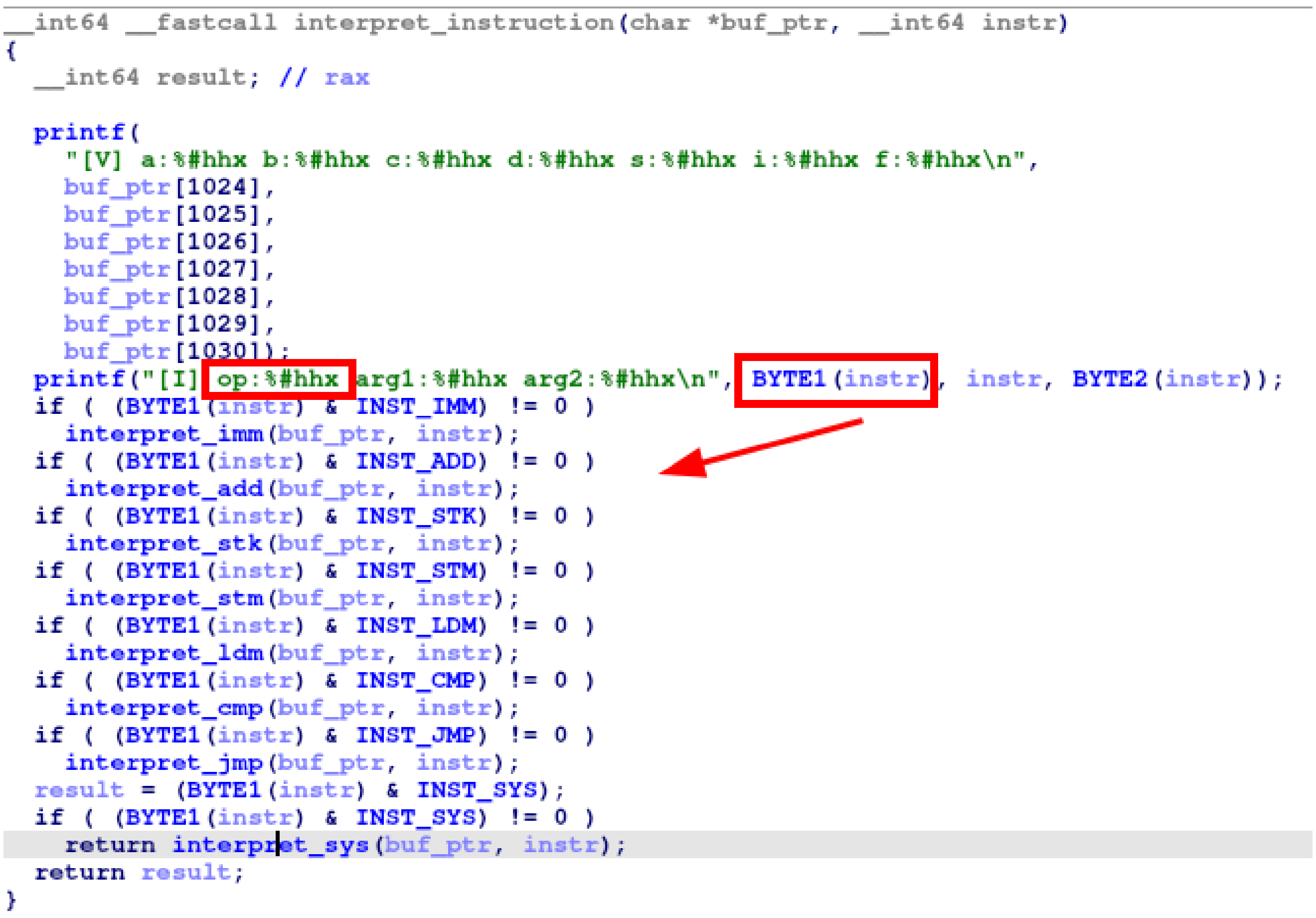
An instruction seems to use just three bytes, even if it is represented by a 8-bytes int.

The syntax of this custom architecture's instructions:

> op arg1 arg2



- byte 0 is the argument number 1
- byte 1 is the operation executed by the cpu
- byte 2 is the argument number 2
  This is a general description, the actual use of the bytes 2 and 0 depends on the operation.



```
__int64 __fastcall interpret_instruction(char *buf_ptr, __int64 instr)
{
  __int64 result; // rax

  printf(
    "[V] a:%#hhx b:%#hhx c:%#hhx d:%#hhx s:%#hhx i:%#hhx f:%#hhx\n",
    buf_ptr[1024],
    buf_ptr[1025],
    buf_ptr[1026],
    buf_ptr[1027],
    buf_ptr[1028],
    buf_ptr[1029],
    buf_ptr[1030]);
  printf("[I] op:%#hhx arg1:%#hhx arg2:%#hhx\n", BYTE1(instr), instr, BYTE2(instr));
  if ( (BYTE1(instr) & INST_IMM) != 0 )
    interpret_imm(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_ADD) != 0 )
    interpret_add(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_STK) != 0 )
    interpret_stk(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_STM) != 0 )
    interpret_stm(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_LDM) != 0 )
    interpret_ldm(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_CMP) != 0 )
    interpret_cmp(buf_ptr, instr);
  if ( (BYTE1(instr) & INST_JMP) != 0 )
    interpret_jmp(buf_ptr, instr);
  result = (BYTE1(instr) & INST_SYS);
  if ( (BYTE1(instr) & INST_SYS) != 0 )
    return interpret_sys(buf_ptr, instr);
  return result;
}
```

We can see that the byte 1 is used to choose what sub-interpretation module to call

Yan85 Architecture

Syscalls

Open(syscall_index_open, reg)

```
int linux_open(const char* filename, int oflag)

yan_open = linux_open(&mem[REG_A], REG_B, REG_C)
```

Read_Memory(syscall_index_read_mem, reg)

```
int linux_read(int fd, void* buf, size_t n_bytes)

yan_read_memory = linux_read(REG_A, &MEM[REG_B], REG_C)
```

Read_code(syscall_index_read_code, reg)

```
int linux_read(int fd, void* buf, size_t n_bytes)

yan_read_code = linux_read(REG_A, &BUF[3*REG_B], REG_C)
```

write(syscall_index_read, reg)

```
int linux_write(int fd, const void* buf, size_t n_bytes)

yan_read_memory = linux_write(REG_A, &MEM[REG_B], REG_C)
```