目录

CPU模拟利器：Unicorn

- 最新版本：`v2.1`
- 更新时间：`20230901`

简介

介绍如何用Unicorn去模拟CPU去执行函数的二进制代码而得到结果。先给出Unicorn的概览；再介绍如何初始Unicorn，包括下载和安装以及运行测试代码确保环境无误。接着详细介绍如何使用Unicorn，yo尤其是概述Unicorn核心思路，接着介绍背景知识，包括程序运行的本质，涉及到CPU的核心逻辑和搞懂什么是指令；以及内存布局相关内容，包括什么是内存布局、内存地址空间范围、典型的内存布局；以及字节序endian。接着介绍运行前的准备，包括设置代码和其他内容。其他内容包含函数参数、相关数据、Stack栈、heap堆；接着介绍运行中的内容，包括如何开始运行，如何调试逻辑，主要是hook，包括hook代码和指令、hook内存、hook异常等内容；以及打印日志，包括优化日志输出。然后是运行后的内容，包括停止运行的逻辑，以及获取结果。接着整理了大量的经验和心得，包括常见的错误，比如UC_ERR_MAP、UC_ERR_WRITE_UNMAPPED等，其中涉及到br间接跳转去混淆相关内容；其他心得还包括手动修改指令、内存读取和写入时的数值转换、如何调用其他子函数、以及具体的模拟malloc、free、vm_deallocate等具体函数实现；以及用到Unicorn的showcase。然后加上一些实际案例，比如模拟akd函数symbol2575和其他一些示例代码；最后附录整理一些Unicorn文档和资料以及Unicorn的部分核心代码。最后贴出引用资料。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/cpu_emulator_unicorn: CPU模拟利器：Unicorn

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- CPU模拟利器：Unicorn book.crifan.org
- CPU模拟利器：Unicorn crifan.github.io

离线下载阅读

- CPU模拟利器：Unicorn PDF
- CPU模拟利器：Unicorn ePub
- CPU模拟利器：Unicorn Mobi

版权和用途说明

此电子书教程的全部内容，如无特别说明，均为本人原创。其中部分内容参考自网络，均已备注了出处。如发现有侵权，请通过邮箱联系我 `admin 艾特 crifan.com`，我会尽快删除。谢谢合作。

各种技术类教程，仅作为学习和研究使用。请勿用于任何非法用途。如有非法用途，均与本人无关。

鸣谢

感谢我的老婆**陈雪**的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

其他

作者的其他电子书

本人 `crifan` 还写了其他 `150+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

关于作者

关于作者更多介绍，详见：

关于CrifanLi李茂 – 在路上

Unicorn概览

- `Unicorn`
  - 一句话描述：终极CPU模拟器=The ultimate CPU emulator
  - 也称为：`Unicorn引擎`
  - 是什么
    - CPU模拟器
    - 跨平台跨语言的CPU仿真库
  - 功能：支持多种架构
    - ARM, AArch64(=ARM64=ARMv8), M68K, MIPS, Sparc, PowerPC, RISC-V, S390x(=SystemZ), TriCore, X86/x86_64
  - 典型用途：模拟调试，用逆向的静态分析难以逆向出代码逻辑的函数、代码、（so等）文件
  - 功能特点
    - API好用 == 干净/简单/轻量级/直观的架构中立的API
    - 支持多种语言 == 以纯C语言实现，可绑定多种其他语言：Pharo、Crystal、Clojure、VB、Perl、Rust、Haskell、Ruby、Python、Java、Go、D、Lua、JS、.NET、Delphi/Pascal、MSVC
    - 支持多种主机系统 == 原生支持 Windows 和 *nix*（已确认 *macOS*、*Linux*、*Android*、BSD 和 Solaris）
    - 通过使用JIT技术实现高性能
    - 支持各种级别的细粒度检测
    - 线程安全的设计
  - 具体实现
    - 底层是依赖于/参考自QEMU的
      - 依赖的QEMU的版本
        - `< 2.0.0 Unicorn`：基于 `QEMU v2.2.1`
        - `>= 2.0.0 Unicorn`：基于 `QEMU v5.0.1`

Unicorn对比QEMU

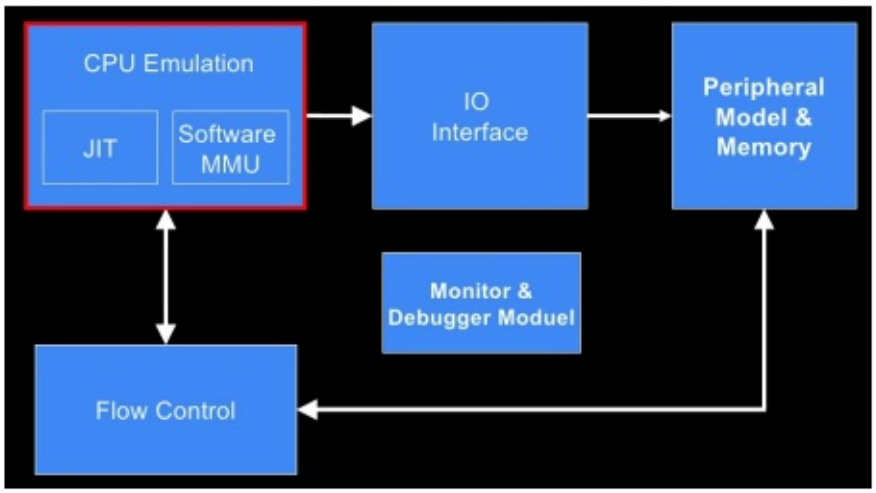
- 类比
  - 如果说QEMU像汽车，那Unicorn就像（汽车最核心的）发动机
- 对比
  - QEMU：模拟整个电脑（CPU+其他：内存，设备、BIOS、固件等等）

Introduction on Qemu

Qemu project

- Open source project (GPL license) on system emulator: http://www.qemu.org
- Huge community & highly active
- Multi-arch
  - X86, Arm, Arm64, Mips, PowerPC, Sparc, etc (18 architectures)
- Multi-platform
  - Compile on *nix + cross-compile for Windows

Qemu architecture



Courtesy of cmchao

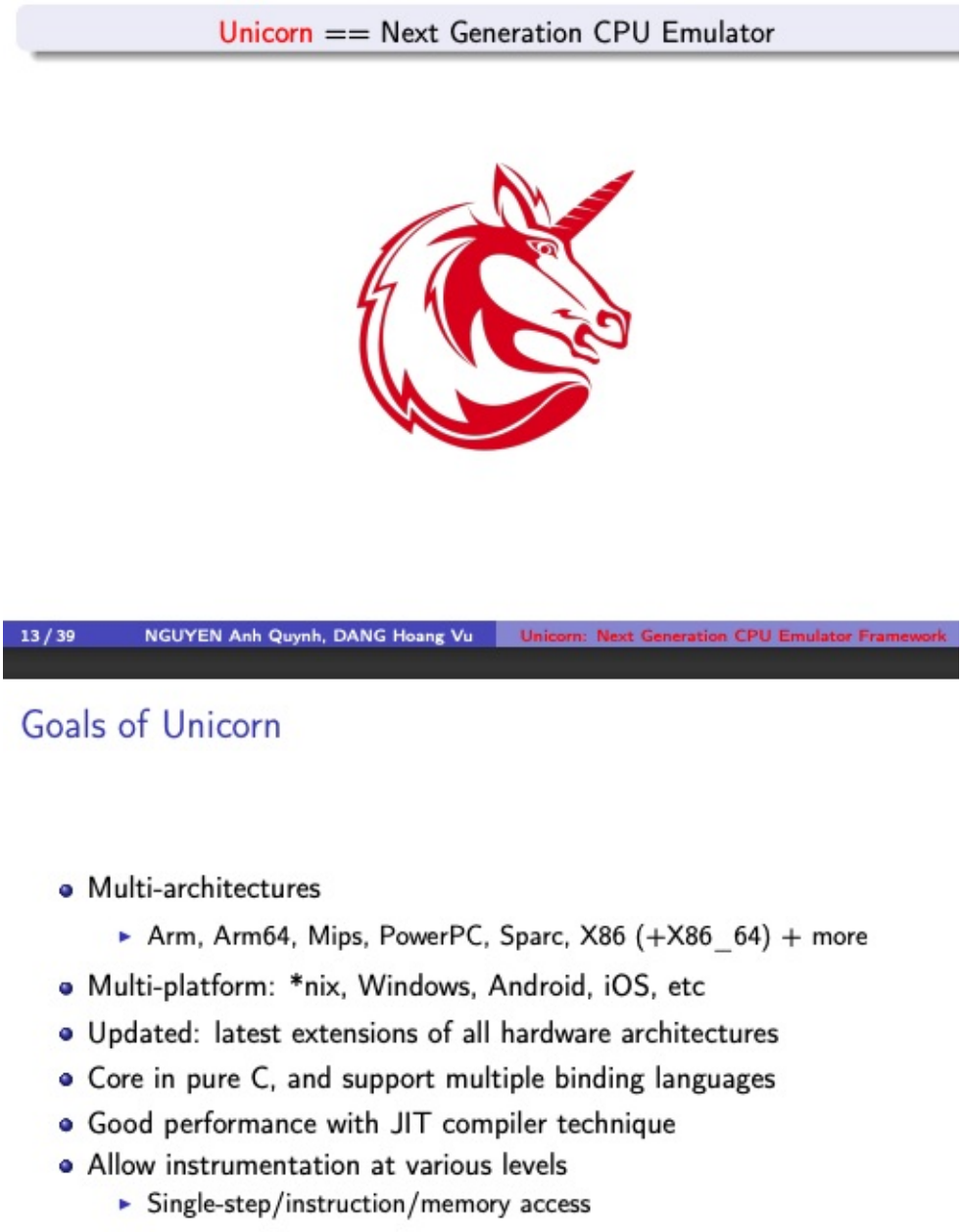
- unicorn：只模拟CPU

Existing CPU emulators

Features	libemu	PyEmu	IDA-x86emu	libCPU	Dream
Multi-arch	X	X	X	X [1]	✓
Updated	X	X	X	X	✓
Independent	X [2]	X [3]	X [4]	✓	✓
JIT	X	X	X	✓	✓

- Multi-arch: existing tools only support X86
- Updated: existing tools do not supports X86_64

[1] Possible by design, but nothing actually works
[2] Focus only on detecting Windows shellcode
[3] Python only
[4] For IDA only



- 结论：
  - Unicorn = QEMU的CPU + 额外优化
    - 即：unicorn是把QEMU中模拟CPU的部分提取出来后，加上其他各种优化
    - 相对QEMU来说，Unicorn有更多的优势
      - 是个框架，支持当做库去调用
      - 用起来更加灵活
      - 支持更多调试手段
      - 线程安全=支持同时多个线程使用
      - 支持更多其他语言接口=bindings
      - 更加轻量级
      - 相对更加安全（和QEMU比减少了攻击面）

初始化

Unicorn的开发环境初始化，包括下载安装Unicorn和运行测试代码，确保环境OK。

下载安装

- 安装Unicorn的核心逻辑
  - Mac中安装unicorn
    - 先安装unicorn本身

```
brew install unicorn
```

      - 安装后：
        - 只有lib库，没有unicorn二进制
          - 库的位置
            - `/usr/local/opt/unicorn/lib/`
        - 由于homebrew把库安装到其自己的目录，所以为了后续程序能找到并加载到unicorn的库，需要去加上对应的环境变量

```
export DYLD_LIBRARY_PATH=/usr/local/opt/unicorn/lib/:$DYLD_LIBRARY_PATH
```

    - 再去安装binding
      - Python的binding

```
pip install unicorn
```

注：如果期间报错，缺少一些依赖的工具，再去分别安装：

- cmake
  - 从官网下载到cmake-3.26.4-macos-universal.dmg，下载后双击，拖动 `CMake` 到应用程序即可
  - 把cmake二进制路径加到PATH，确保命令行中能用cmake
    - 编辑启动脚本 `vi ~/.zshrc` ，加上： `export PATH="/Applications/CMake.app/Contents/bin":"$PATH"` ，即可。
- pkg-config
  - `brew install pkg-config`

运行测试代码

安装完毕后，继续去运行Unicorn的测试代码，确保代码能正常运行和输出期望的值，以确保环境正常。

比如此处运行Python的ARM的测试代码：

- unicorn/sample_arm.py at master · unicorn-engine/unicorn · GitHub

```
#!/usr/bin/env python
# Sample code for ARM of Unicorn. Nguyen Anh Quynh <aquynh@gmail.com>
# Python sample ported by Loi Anh Tuan <loianhtuan@gmail.com>

from __future__ import print_function
from unicorn import *
from unicorn.arm_const import *

# code to be emulated
ARM_CODE    = b"\x37\x00\xa0\xe3\x03\x10\x42\xe0" # mov r0, #0x37; sub r1, r2, r3
THUMB_CODE = b"\x83\xb0" # sub    sp, #0xc
# memory address where emulation starts
ADDRESS     = 0x10000

# callback for tracing basic blocks
def hook_block(uc, address, size, user_data):
    print(">>> Tracing basic block at 0x%x, block size = 0x%x" %(address, size))

# callback for tracing instructions
def hook_code(uc, address, size, user_data):
    print(">>> Tracing instruction at 0x%x, instruction size = 0x%x" %(address, size))

# Test ARM
def test_arm():
    print("Emulate ARM code")
    try:
        # Initialize emulator in ARM mode
        mu = Uc(UC_ARCH_ARM, UC_MODE_ARM)

        # map 2MB memory for this emulation
        mu.mem_map(ADDRESS, 2 * 1024 * 1024)

        # write machine code to be emulated to memory
        mu.mem_write(ADDRESS, ARM_CODE)

        # initialize machine registers
        mu.reg_write(UC_ARM_REG_R0, 0x1234)
        mu.reg_write(UC_ARM_REG_R2, 0x6789)
        mu.reg_write(UC_ARM_REG_R3, 0x3333)
        mu.reg_write(UC_ARM_REG_APSR, 0xFFFFFFFF) #All application flags turned on

        # tracing all basic blocks with customized callback
        mu.hook_add(UC_HOOK_BLOCK, hook_block)

        # tracing one instruction at ADDRESS with customized callback
        mu.hook_add(UC_HOOK_CODE, hook_code, begin=ADDRESS, end=ADDRESS)
```

```
        # emulate machine code in infinite time
        mu.emu_start(ADDRESS, ADDRESS + len(ARM_CODE))

        # now print out some registers
        print(">>> Emulation done. Below is the CPU context")

        r0 = mu.reg_read(UC_ARM_REG_R0)
        r1 = mu.reg_read(UC_ARM_REG_R1)
        print(">>> R0 = 0x%x" %r0)
        print(">>> R1 = 0x%x" %r1)

    except UcError as e:
        print("ERROR: %s" % e)

def test_thumb():
    print("Emulate THUMB code")
    try:
        # Initialize emulator in thumb mode
        mu = Uc(UC_ARCH_ARM, UC_MODE_THUMB)

        # map 2MB memory for this emulation
        mu.mem_map(ADDRESS, 2 * 1024 * 1024)

        # write machine code to be emulated to memory
        mu.mem_write(ADDRESS, THUMB_CODE)

        # initialize machine registers
        mu.reg_write(UC_ARM_REG_SP, 0x1234)

        # tracing all basic blocks with customized callback
        mu.hook_add(UC_HOOK_BLOCK, hook_block)

        # tracing all instructions with customized callback
        mu.hook_add(UC_HOOK_CODE, hook_code)

        # emulate machine code in infinite time
        # Note we start at ADDRESS | 1 to indicate THUMB mode.
        mu.emu_start(ADDRESS | 1, ADDRESS + len(THUMB_CODE))

        # now print out some registers
        print(">>> Emulation done. Below is the CPU context")

        sp = mu.reg_read(UC_ARM_REG_SP)
        print(">>> SP = 0x%x" %sp)

    except UcError as e:
        print("ERROR: %s" % e)

def test_read_sctlr():
    print("Read SCTLR")
    try:
        # Initialize emulator in thumb mode
        mu = Uc(UC_ARCH_ARM, UC_MODE_ARM)

        # Read SCTLR
```

```
        # cp = 15
        # is64 = 0
        # sec = 0
        # crn = 1
        # crm = 0
        # opc1 = 0
        # opc2 = 0
        val = mu.reg_read(UC_ARM_REG_CP_REG, (15, 0, 0, 1, 0, 0, 0))
        print(">>> SCTLR = 0x%x" % val)

    except UcError as e:
        print("ERROR: %s" % e)

if __name__ == '__main__':
    test_arm()
    print("=" * 26)
    test_thumb()
    print("=" * 26)
    test_read_sctlr()
```

此处输出期望的结果：

```
crifan@licrifandeMacBook-Pro ~/dev/dev_src/ios_reverse/unicorn/unicorn  master  pyt
hon bindings/python/sample_arm.py
Emulate ARM code
>>> Tracing basic block at 0x10000, block size = 0x8
>>> Tracing instruction at 0x10000, instruction size = 0x4
>>> Emulation done. Below is the CPU context
>>> R0 = 0x37
>>> R1 = 0x3456
==========================
Emulate THUMB code
>>> Tracing basic block at 0x10000, block size = 0x2
>>> Tracing instruction at 0x10000, instruction size = 0x2
>>> Emulation done. Below is the CPU context
>>> SP = 0x1228
==========================
Read SCTLR
>>> SCTLR = 0xc50078
```

即表示Unicorn环境正常可用。

如何使用

接下来，详细解释，如何在真实的开发环境，尤其是iOS逆向期间，去使用Unicorn模拟代码运行。

概述Unicorn核心思路

使用Unicorn去模拟代码执行的核心思路是：

- 背景知识
  - 先要具备相关技术背景知识，至少包括
    - 对于程序运行本质（读取指令，运行指令）有所了解
    - 对基本的典型的内存布局有所了解
- 运行前
  - 先要准备好要运行的代码
    - 往往是对应的二进制文件
      - 比如用lldb调试iOS逆向的iPhone中程序期间，导出的某个函数的全部或部分的代码
  - 稍微复杂点的情况，还要准备其他相关内容
    - 设置好要模拟的传入函数的参数
    - 给特定内存位置写入对应的数据
    - 如果程序内部跳转调用其他子函数，则还要设置好：Stack栈
    - 如果期间涉及到malloc等内存分配，则还要准备好：Heap堆
- 运行中
  - 主要就是调用 `emu_start` 触发开始模拟
  - 期间更多的是，要调试搞懂代码逻辑或者查看对应的寄存器或内存的值
    - 所以往往要用对应的手段去调试
      - 典型都有hook机制：hook代码（甚至hook特定指令）、hook内存、hook异常
      - 以及用好日志打印，其中有很多可以优化的地方
      - 包括可能利用Capstone去查看当前正在运行的是什么指令
- 运行后
  - 常常是判断指令是 `ret` 时调用 `emu_stop` 而停止模拟运行
  - 然后再去从返回的寄存器或特定内存地址，获取程序模拟的最终的输出结果

下面详细解释具体过程。

背景知识

想要能玩转Unicorn，模拟代码运行，前提是：对于相关涉及到的背景知识，基础知识，有一定的了解。

此处至少包括：

- 程序运行的本质
- 典型的内存布局

下面详细解释：

程序运行的本质

此处所说的，程序运行的本质，主要指CPU模拟指令运行方面的底层实现原理和机制。

由于此部分内容较多，不适合完全铺开介绍，所以此处尽量言简意赅，把问题解释清楚。

CPU的核心逻辑

CPU的运行，本质上就是：从内存中读取指令，并运行指令（包括输出结果，到对应内存地址或寄存器）

Unicorn模拟CPU的核心逻辑

此处Unicorn要模拟的是CPU的运行。所以也就（只）是，把代码放到对应的地址上，Unicorn开始运行，去对应地址：**读取指令**，（解析并）**执行指令**，即可。

而解析和运行该指令的结果，往往是，本身就是，写入计算后的结果到对应的寄存器或内存而已。

拿最基础的常见的例子来说：让CPU去计算 `2` + `3`，则底层逻辑（你暂时无需知道底层的具体的汇编指令，而只需要知道），肯定就是类似的这种步骤：

把数值 `2` 放到一个寄存器A中，把 `3` 放到另外一个寄存器B中，然后执行寄存器A加上寄存器B，然后计算的结果，保存到寄存器A中，或者另外写入到寄存器C中，甚至写入到某个内存地址，供后续读取和使用

如此，Unicorn模拟的就只是，CPU的指令的读取、解析、运行、输出结果的过程，而已。

而在指令执行期间的所需要的其他内容，比如后续会涉及到的函数参数、Stack栈、Heap堆等等，则都是为了：确保Unicorn模拟CPU的结果，和真实的代码执行的结果，要（完全）一致，才有价值，才能真正得到的希望的输出的结果。

什么是：指令=数据=操作码=二进制

先明白一下常说的概念和逻辑：

- `指令` = `instruction` == `代码` = `code` == `操作码` = `opcode` == `二进制（数据）` == `binary（data）`

分别解释大概的含义：

- `指令` = `instruction`
  - 往往是底层会汇编代码级别，才会用到这个名词，指令。
  - 比如：ARM汇编指令
- `代码` = `code`
  - 往往是高级语言常用到的名字
  - 比如：iOS的ObjC的代码
- `操作码` = `opcode` = `operation code`
  - **= 操作机器的代码**
    - 机器 = CPU -》 操作CPU（让其按照你的预期）去工作的代码
  - 往往是底层某个架构对应的指令集中才会用到这个名字，opcode
  - 比如：ARM指令集特定的指令的opcode
- `二进制（数据）` = `binary（data）`
  - 最底层的硬件中所保存的数据，都是0101的形式的二进制数据

综合起来就是：

你的上层语言的 `代码`（比如iOS的ObjC），经过编译，最终底层生成的都是 `二进制数据`

而这些二进制数据，对应着（比如ARM指令集中的） `指令`，也就是对应指令的 `操作码`（可以通过查询ARM手册，从二进制的0101等数据，慢慢反推出对应是具体什么指令）

内存布局

先说原因：使用Unicorn之前，要搞懂内存布局，是因为：

只有所有的东西都放到合适的（该放的）位置后，代码才能正常（按照你的预期，模拟实际的情况去）运行，才能得到你要的输出结果

下面接着介绍内存布局相关细节：

什么是内存布局

内存布局，指的是：

- 你把要放置的东西，具体放哪里了

而此处的：

- 要放置的东西
  - 主要指的是：`代码` = `code`
    - 详见后续章节：设置代码
  - 其他往往也涉及到
    - `数据` = `data`
    - 其他 = 能让程序正常运行起来的各种配合的环境
      - `Stack` 栈
        - 详见后续章节：Stack栈
      - `Heap` 堆
        - 详见后续章节：Heap堆
      - 有时候还要
        - 给特定内存写入特定值
          - 详见后续章节：相关数据
      - 等等
- 放哪里 中的 哪里 指的是：`内存` = `memory`
- 具体放哪里了 中的 具体 指的是：放到**内存**中的哪个**范围**了
  - 往往涉及到：`起始地址` + 这段的 `空间大小` = `结束地址`

内存地址空间范围

再来说，被放置的地方，此处叫做：内存，以及内存的具体的范围。

此处的Unicorn所模拟的内存，指的是 `虚拟内存` 。

虚拟内存的范围即空间大小，可以简单的理解为：

- 32位：2^32= `4GB`
- 64位：
  - 理论上：2^64= `16777216TB`
  - 实际上：一般也支持 >= `256TB` 的实际物理内存大小

-》所以，Unicorn模拟时，可以认为：

可用的内存空间大小：32位是 `4GB` ，64位是无限可用的，比如远不止 `256TB`

典型的内存布局

搞懂了什么是内存布局，再来聊聊，典型的常见的内存布局。

常见的内存布局，内容很多。

用如下图（别人画的，我加了点内容），大概先解释一下，好让大家有个概念：

- （程序的）典型的内存布局

  -

内存布局举例

下面，通过具体例子，来解释，内存布局的细节含义：

官网示例代码中的内存布局

已我们前面运行测试代码中用到的官网示例代码unicorn/sample_arm.py来说，其中就是：

- 要模拟执行的代码code
  - 具体是什么：`ARM_CODE = b"\x37\x00\xa0\xe3\x03\x10\x42\xe0" # mov r0, #0x37; sub`

`r1, r2, r3`

- 具体放到内存什么地方=范围：`ADDRESS = 0x10000`

意思是：

把代码 `\x37\x00\xa0\xe3\x03\x10\x42\xe0` 放到内存地址为 `0x10000` 开始的地方

就没了。是的。即没有我们提到的，函数参数，也没有Stack、Heap，特定地址写入特定值等内容

因为，此处要模拟的，只是一个代码片段，不是一个完整的函数。

此处只是官网示例的代码，用于演示运行，确保Unicorn模拟环境正常而已。所以没有用完整的函数，只是一小段代码而已。

此时的内存布局，可以理解为：

```
Mapped memory: Code     [0x0000000000010000 - 4GB]
```

或：

```
Mapped memory: Code     [0x0000000000010000 - 0x0000000000010008]
```

其中：0x0000000000010008 - 0x0000000000010000 = 8 = 上述代码的大小=字节个数

实际例子中的内存布局

之前自己某个Unicorn模拟函数的代码，中的内存布局：

早期是：

```
Mapped memory: Code     [0x0000000000010000-0x0000000000410000]
Mapped memory: Libc     [0x0000000000500000-0x0000000000580000]
Mapped memory: Heap     [0x0000000000600000-0x0000000000700000]
Mapped memory: Stack    [0x0000000000700000-0x0000000000800000]
Mapped memory: Args     [0x0000000000800000-0x0000000000810000]
```

经过多次优化，最后是：

```
Mapped memory: Code     [0x00010000-0x00410000]
                          [0x00010000-0x000124C8] func: ___lldb_unnamed_symbol2575$$akd
                          [0x00031220-0x00033450]   fix br err: x9SmallOffset
                          [0x00068020-0x00069B80]   fix br err: x10AbsFuncAddrWithOffset
                          [0x00069B88-0x00069B90]   emulateFree jump
                          [0x00069BC0-0x00069BC8]   emulateAkdFunc2567 jump
                          [0x00069BD8-0x00069BE0]   emulateMalloc jump
                          [0x00069BE8-0x00069BF0]   line 7392 jump
                          [0x00069C08-0x00069C10]   emulateDemalloc jump
                          [0x00200000-0x00200004] func: emulateMalloc
                          [0x00220000-0x00220004] func: emulateFree
                          [0x00280000-0x00280004] func: emulateAkdFunc2567
Mapped memory: Libc     [0x00500000-0x00580000]
Mapped memory: Heap     [0x00600000-0x00700000]
```

```
Mapped memory: Stack    [0x00700000-0x00800000]
Mapped memory: Args     [0x00800000-0x00810000]
```

具体布局的详细解释，详见后续章节：

模拟akd函数symbol2575

其他一些内存布局

afl-unicorn: Fuzzing Arbitrary Binary Code | by Nathan Voss | HackerNoon.com | Medium

中的Unicorn的内存布局：

```
# Memory map for the code to be tested
CODE_ADDRESS  = 0x00100000 # Arbitrary address where code to test will be loaded
CODE_SIZE_MAX  = 0x00010000 # Max size for the code (64kb)
STACK_ADDRESS  = 0x00200000 # Address of the stack (arbitrarily chosen)
STACK_SIZE  = 0x00010000 # Size of the stack (arbitrarily chosen)
DATA_ADDRESS  = 0x00300000 # Address where mutated data will be placed
DATA_SIZE_MAX  = 0x00010000 # Maximum allowable size of mutated data
```

另外，其中还有一些更加细节的内容，比如

- 初始化寄存器 `__load_registers`
- 映射内存 的 同时设置读写属性 -》用于内存保护

可供参考。

以及：

afl-unicorn/unicorn_loader.py at master · Battelle/afl-unicorn · GitHub

中的：

- `__map_segments`
  - `__map_segment`

字节序endian

写入内容（代码，数据）到内存中之前，要注意先确认 `字节序` = `endian` 是 `大端` 还是 `小端`

大端big endian vs 小端 little endian

- 文字
  - 大小端：0x12345678
    - 大端：高字节放低地址
      - 0x00-0x07：12 34 56 78
    - 小端：高字节 放 高地址
      - 0x00-0x07：78 56 34 12
- 图

Unicorn中endian设置

ARM中，默认是 `小端` = `UC_MODE_LITTLE_ENDIAN`

除非特殊需要，才会设置为 `大端` = `UC_MODE_BIG_ENDIAN`

比如，官网例子给出了演示如何设置大端的用法：

unicorn/sample_arm64eb.py at master · unicorn-engine/unicorn · GitHub

```
    print("Emulate ARM64 Big-Endian code")
    try:
```

```
        # Initialize emulator in ARM mode
        mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM | UC_MODE_BIG_ENDIAN)
```

运行前

在能用Unicorn模拟运行你的代码之前，要准备好相关的内容，即，内存中要 设置=存放 好相关的东西。

下面详细介绍具体要设置哪些东西。

设置代码

如果要让CPU能运行你的指令，你要先去把你的代码，放到内存中，供CPU读取。

所以，第一个优先要放到内存中的，就是：`代码`。

- 注：代码==指令==二进制==opcode

而你把代码放到具体哪个位置，就是内存布局中，代码的地址空间范围。

举例

官网示例

官网示例 unicorn/sample_arm.py 中的：

```
# code to be emulated
ARM_CODE    = b"\x37\x00\xa0\xe3\x03\x10\x42\xe0" # mov r0, #0x37; sub r1, r2, r3
```

就是对应的：要运行的代码

被翻译成ARM汇编的话，对应的内容就是，代码后面的注释的部分

```
mov r0, #0x37
sub r1, r2, r3
```

怎么看出来上面的二进制就是上面的ARM汇编代码的?

如之前：什么是指令 所解释的：

此处的 二进制 == ARM的指令

而具体如何看出来：

- 二进制：`\x37\x00\xa0\xe3\x03\x10\x42\xe0`

就是对应的ARM汇编代码：

```
mov r0, #0x37
sub r1, r2, r3
```

则可以：

- 要么是自己手动去分析，把二进制一点点根据对应的bit位，手动分析出对应的汇编指令
  - 详见：最流行汇编语言：ARM (crifan.org)
- 要么是，可以借助工具
  - 在线网站：把二进制转换成ARM汇编指令
    - Online ARM to HEX Converter (armconverter.com)

此处以在线网站

Online ARM to HEX Converter (armconverter.com)

举例来说明：

- ARM汇编指令 `mov r0, #0x37` 对应的ARM的 `指令` = `二进制` 是： `3700A0E3` = `\x37\x00\xA0\xE3`
  -
- ARM汇编指令 `sub r1, r2, r3` 对应的ARM的 `指令` = `二进制` 是： `031042E0` == `\x03\x10\x42\xE0`
  -

-》把 `\x37\x00\xA0\xE3` 和 `\x03\x10\x42\xE0` 加起来，就是此处的二进制： `\x37\x00\xa0\xe3\x03\x10\x42\xe0` 了。

自己的实例

自己在后续示例

模拟akd函数symbol2575

中，其中的设置代码的部分就是：

```
def readBinFileBytes(inputFilePath):
    fileBytes = None
    with open(inputFilePath, "rb") as f:
        fileBytes = f.read()
    return fileBytes

# for arm64: ___lldb_unnamed_symbol2575$$akd
akd_symbol2575_FilePath = "input/akd_getIDMSRoutingInfo/arm64/akd_arm64_symbol2575.bin"
logging.info("akd_symbol2575_FilePath=%s", akd_symbol2575_FilePath)
ARM64_CODE_akd_symbol2575 = readBinFileBytes(akd_symbol2575_FilePath) # b'\xff\xc3\x03\
xd1\xfco\t\xa9\xfag\n\xa9\xf8_\x0b\xa9\xf6W\x0c\xa9\xf4O
```

就是从，输入文件

- `input/akd_getIDMSRoutingInfo/arm64/akd_arm64_symbol2575.bin`

  -

中，读取对应代码=二进制数据，供后续模拟运行的。

而该文件的数据是来自于，lldb调试期间，从内存中导出的：

```
(lldb) memory read --binary --force --outfile /Users/crifan/dev/tmp/lldb_mem_dump/akd_a
rm64_symbol2575.bin 0x10485d98c 0x1048600dc
10064 bytes written to '/Users/crifan/dev/tmp/lldb_mem_dump/akd_arm64_symbol2575.bin'
```

其中 `0x10485d98c` 和 `0x1048600dc` 是该函数的起始地址和结束地址。

2023-06-07 23:00:39

设置其他

而能让CPU顺利模拟运行你的代码之外，对于实际情况中，比如稍微复杂一点的代码，往往还有会涉及到其他一些内容：

- 函数的参数
- Stack栈
- heap堆
- 事先要写入特定内存的值

下面分别介绍一下：

函数参数

很多要模拟的函数，往往是有一些参数的，所以在模拟之前，要先把参数写入对应寄存器或内存，供模拟执行用。

比如，你模拟 `add(int a, int b)` 的二进制代码执行的话，就要把 `a` 和 `b` 的值先写好，比如放到ARM的寄存器 `x0` 和 `x1` 中，供模拟调用。

实例

以 模拟akd函数symbol2575 为例，当时此处函数要传入的参数值，分别是：

```
#------------------- Args -------------------
# memory address for arguments
ARGS_ADDRESS = 8 * 1024 * 1024
ARGS_SIZE =  0x10000

# init args value
ARG_routingInfoPtr = ARGS_ADDRESS
ARG_DSID = 0xfffffffffffffffe
'''
        # for current arm64 ___lldb_unnamed_symbol2575$$akd =====
        mu.reg_write(UC_ARM64_REG_X0, ARG_DSID)
        mu.reg_write(UC_ARM64_REG_X1, ARG_routingInfoPtr)
```

- 第一个参数的存放位置：ARM64中的 `x0` 寄存器
  - 存放的值：`0xfffffffffffffffe`
    - 是所要模拟的函数，（用 `Xcode` / `lldb` / `Frida` ）调试出来的真实函数调用时传入的值
- 第二个参数的存放位置：ARM64中的 `x1` 寄存器
  - 存放的值：此处也是调试出真实函数的逻辑，此处是特殊的情况，传入一个指针，该指针用于保存最终要返回的值
    - 所以此处设置一个地址，用于保存后续的返回值
      - 而此处的地址，则选择了内存布局中，高地址部分的 `8 * 1024 * 1024 = 8MB` 的位置
        - 只要和别处不冲突，任何地址都可以

注：关于其中的，ARM寄存器保存参数的逻辑，属于：

- ARM的函数调用规范
  - 概述：
    - 64位的ARM中，函数参数个数不超过8个，分别保存到 `X0` ~ `X7` 中
      - 对比：32位的ARM时，函数参数个数不超过4个，分别保存到 `X0` ~ `X3` 中
    - 超出的函数参数，则放到Stack栈中
  - 详见：调用规范 · 最流行汇编语言：ARM (crifan.org)

相关数据

其他一些特殊情况中，要给特定内存地址写入特定地址，供后续代码模拟时调用。

一般普通的函数模拟，往往无需此过程。

自己的实例

此处以后续的 模拟akd函数symbol2575 为例，来解释特殊的情况：

由于函数 `___lldb_unnamed_symbol2575$$akd` 做了特殊的反调试处理：代码中很多 `BR` 间接跳转，导致需要写入特定内存地址中，特定的值，供后续代码运行时读取，才能正常跳转到对应的行的代码，继续正确运行。

此时就涉及到，要向特定内存地址，写入特定的值。

- 要向什么地址？写入具体什么值？

比如某次调试报错时：

```
=== 0x00011130  +4400 : 36 D9 68 F8  -> ldr     x22, [x9, w8, sxtw #3]
 << Memory READ at 0x69C18, size=8, rawValueLittleEndian=0x0000000000000000, pc=0x11130
```

涉及到要：

- 要写入的地址，就是：`0x69C18`

而要写入的值： 则需要（用工具 `Xcode` / `lldb` / `Frida` 去）调试去真正的函数执行期间的值，此处调试出是：

- 要写入的值：`0x0000000000078dfa`

然后就可以：

- 向要写入的地址：`0x69C18`
  - 写入具体的值：`0x0000000000078dfa`
    - 且占用地址空间大小是：8字节=64bit的值

如此，即可去调用自己优化后的代码：

向特定内存地址，写入对应字节大小的特定的值

```
uc = None

def writeMemory(memAddr, newValue, byteLen):
    """
        for ARM64 little endian, write new value into memory address
        memAddr: memory address to write
        newValue: value to write
        byteLen: 4 / 8
    """
```

```
    global uc

    valueFormat = "0x%016X" if byteLen == 8 else "0x%08X"
    if isinstance(newValue, bytes):
        logging.info("writeMemory: memAddr=0x%X, newValue=0x%s, byteLen=%d", memAddr, n
ewValue.hex(), byteLen)
        newValueBytes = newValue
    else:
        valueStr = valueFormat % newValue
        logging.info("writeMemory: memAddr=0x%X, newValue=%s, byteLen=%d", memAddr, val
ueStr, byteLen)
        newValueBytes = newValue.to_bytes(byteLen, "little")
    uc.mem_write(memAddr, newValueBytes)
    logging.info(" >> has write newValueBytes=%s to address=0x%X", newValueBytes, memAd
dr)

    # # for debug: verify write is OK or not
    # readoutValue = uc.mem_read(memAddr, byteLen)
    # logging.info("for address 0x%X, readoutValue hex=0x%s", memAddr, readoutValue.hex
()))
    # # logging.info("readoutValue hexlify=%b", binascii.hexlify(readoutValue))
    # readoutValueLong = int.from_bytes(readoutValue, "little", signed=False)
    # logging.info("readoutValueLong=0x%x", readoutValueLong)
    # # if readoutValue == newValue:
    # if readoutValueLong == newValue:
    #     logging.info("=== Write and read back OK")
    # else:
    #     logging.info("!!! Write and read back Failed")


def emulate_akd_arm64_symbol2575():
    global uc, ucHeap

    mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM + UC_MODE_LITTLE_ENDIAN)
    uc = mu

    ...

    writeMemory(0x69C18, 0x0000000000078dfa, 8) # <+4400>: 36 D9 68 F8  -> ldr     x22,
 [x9, w8, sxtw #3]
```

即可实现，模拟运行时，读取出正确的写入的raw value：

```
=== 0x00011130 +4400 : 36 D9 68 F8  -> ldr     x22, [x9, w8, sxtw #3]
 << Memory READ at 0x69C18, size=8, rawValueLittleEndian=0xfa8d070000000000, pc=0x11130
```

- 注：此处从内存中读取出来的值是 `0xfa8d070000000000` ，之所以不是（以为的，原先写入的值） `0x0000000000078dfa` ，是因为：此处是ARM64，是little endian=小端，所以内存中原始的值，就是按照 `fa 8d 07 00 00 00 00 00` 存放的。
  - 关于endian的知识，具体详见之前章节：字节序endian

从而使得后续代码逻辑，按照预期的逻辑继续去执行了。

2023-06-13 23:06:17

Stack栈

如果函数A内部调用了其他函数，比如B函数，C函数等，则往往又涉及到函数上下文的切换，底层具体实现就是涉及到Stack栈，在调用之前和之后，会操作Stack，保存 `PC` 、 `LR` 、多个相关寄存器等等，所以，往往在模拟函数运行之前，也要先去设置好Stack。

- Stack栈
  - 本身特性
    - 本质上：是一个线性表
      - 操作栈顶(top)，进行插入或删除
        - 另一端称为栈底bottom
      - 操作原理：后进先出=LIFO=Last In First Out
  - 生成方向=增长方向
    - 理论上支持
      - Full Descending=满递减=由高地址到低地址
      - 空递增？ = 由低地址到高地址
    - 但是基本上都是用
      - Full Descending=满递减=由高地址到低地址

ARM中的Stack栈

ARM中的Stack栈，默认情况下，都是：由高地址到低地址。

举例

自己的实例

以 模拟akd函数symbol2575 为例，其中的Stack栈相关的代码是：

```
#------------------- Stack -------------------
# Stack: from High address to lower address ?
STACK_ADDRESS = 7 * 1024 * 1024
STACK_SIZE = 1 * 1024 * 1024
STACK_HALF_SIZE = (int)(STACK_SIZE / 2)

# STACK_ADDRESS_END = STACK_ADDRESS - STACK_SIZE # 8 * 1024 * 1024
# STACK_SP = STACK_ADDRESS - 0x8 # ARM64: offset 0x8

# STACK_TOP = STACK_ADDRESS + STACK_SIZE
STACK_TOP = STACK_ADDRESS + STACK_HALF_SIZE
STACK_SP = STACK_TOP

...

        # map stack
        mu.mem_map(STACK_ADDRESS, STACK_SIZE)
        logging.info("Mapped memory: Stack\t[0x%08X-0x%08X]", STACK_ADDRESS, STACK_ADDR
ESS + STACK_SIZE)
```

```
'''

        # initialize stack
        # mu.reg_write(UC_ARM64_REG_SP, STACK_ADDRESS)
        mu.reg_write(UC_ARM64_REG_SP, STACK_SP)
```

其中输出的内存布局中的Stack部分就是：

```
Mapped memory: Stack   [0x00700000-0x00800000]
```

其中的：

`mu.reg_write(UC_ARM64_REG_SP, STACK_SP)`

就是真正的Stack初始化的操作，即：设置SP指针，为Stack栈的地址

此处由于是`由高地址到低地址`的Stack，所以最初的`SP`的值是`STACK_TOP`=Stack的最高地址

Heap堆

如果要模拟的函数，内部涉及到申请内存malloc等，则往往也要设置对应的headp堆，用于模拟内存管理提供动态申请和释放内存用。

举例

自己的实例

以 模拟akd函数symbol2575 为例，其中关于Heap堆的代码是：

```
from libs.UnicornSimpleHeap import UnicornSimpleHeap

ucHeap = None

#------------------- Heap --------------------

HEAP_ADDRESS = 6 * 1024 * 1024
HEAP_SIZE = 1 * 1024 * 1024

HEAP_ADDRESS_END = HEAP_ADDRESS + HEAP_SIZE
HEAP_ADDRESS_LAST_BYTE = HEAP_ADDRESS_END - 1

...

# callback for tracing instructions
def hook_code(mu, address, size, user_data):
    global ucHeap
...
    # for emulateMalloc
    # if pc == 0x00200000:
    if pc == EMULATE_MALLOC_CODE_START:
        mallocSize = mu.reg_read(UC_ARM64_REG_X0)
        newAddrPtr = ucHeap.malloc(mallocSize)
        mu.reg_write(UC_ARM64_REG_X0, newAddrPtr)
        logging.info("\temulateMalloc: input x0=0x%x, output ret: 0x%x", mallocSize, ne
wAddrPtr)
        gNoUse = 1

...
        # map heap
        mu.mem_map(HEAP_ADDRESS, HEAP_SIZE)
        logging.info("Mapped memory: Heap\t[0x%08X-0x%08X]", HEAP_ADDRESS, HEAP_ADDRESS
 + HEAP_SIZE)
...
        # init Heap malloc emulation
        ucHeap = UnicornSimpleHeap(uc, HEAP_ADDRESS, HEAP_ADDRESS_LAST_BYTE, debug_print
=True)
```

其中此处是：

要模拟的代码内部，涉及到malloc去分配内容，所以借鉴了别人的代码，更新后，独立出单独的库 `UnicornSimpleHeap` ，然后用于此处模拟调用malloc函数。

而关于Unicorn模拟代码中，调用子函数=调用其他函数的部分，详见：

调用其他子函数

运行中

此处Unicorn在准备好环境后，就可以通过调用 `emu_start` 开始真正的去触发模拟运行代码了。

以及在运行期间，为了调试代码逻辑，还要加上很多额外逻辑，主要是hook各种内容，查看实时的值，是否符合期望。

下面具体解释细节。

开始运行

Unicorn中，真正触发开始模拟代码，是调用函数 `emu_start` 。

emu_start举例

之前官网例子

比如之前：官网测试代码 中的触发开始运行的代码就是：

```
    # emulate machine code in infinite time
    mu.emu_start(ADDRESS, ADDRESS + len(ARM_CODE))
```

自己的实例

以 模拟akd函数symbol2575 为例，其中的触发开始运行的代码是：

```
        # emulate machine code in infinite time
        mu.emu_start(CODE_ADDRESS, CODE_ADDRESS + len(ARM64_CODE_akd_symbol2575))
```

其中：

- 代码起始地址：`CODE_ADDRESS`
  - 最开始映射的代码的最初位置
- 代码结束地址：`CODE_ADDRESS + len(ARM64_CODE_akd_symbol2575)`
  - 映射的代码起始位置，加上对应代码长度后的，结束位置

-》这样可以合理的限定Unicorn要模拟运行的代码指令，而不会额外多去运行（本身是其他数据的）无用的指令

调试逻辑

为了用Unicorn模拟代码运行，调试出我们希望搞懂的函数的逻辑，往往期间需要很多额外的调试内容。

其中主要就是：

- hook
  - hook代码
    - hook特定指令
  - hook内存
  - hook异常
- 打印日志
- 其他常见需求
  - 比如
    - 查看当前运行的是什么指令

下面分别详细介绍。

hook

Unicorn模拟期间，常需要去搞懂底层正在发生的细节，查看对应的寄存器、内存的值等等，此时，就可以用到Unicorn所提供的机制：hook。

其中比较常用的一些hook是：

- hook代码
  - hook特定指令
- hook内存
- hook异常
- hook其他

下面分别解释如何使用和具体效果。

Unicorn支持的全部的hook种类

关于Unicorn支持的hook的全部种类是：

- 指令执行类
  - `UC_HOOK_INTR`
  - `UC_HOOK_INSN`
  - `UC_HOOK_CODE`
  - `UC_HOOK_BLOCK`
- 内存访问类
  - `UC_HOOK_MEM_READ_PROT`
  - `UC_HOOK_MEM_WRITE_PROT`
  - `UC_HOOK_MEM_FETCH_PROT`
  - `UC_HOOK_MEM_READ`
  - `UC_HOOK_MEM_WRITE`
  - `UC_HOOK_MEM_FETCH`
  - `UC_HOOK_MEM_READ_AFTER`
- 异常处理类
  - `UC_HOOK_MEM_READ_UNMAPPED`
  - `UC_HOOK_MEM_WRITE_UNMAPPED`
  - `UC_HOOK_MEM_FETCH_UNMAPPED`
  - `UC_HOOK_INSN_INVALID`
- 其他
  - `UC_HOOK_EDGE_GENERATED`
  - `UC_HOOK_TCG_OPCODE`

可以从官网源码unicorn.h中找到定义：

```
// All type of hooks for uc_hook_add() API.
typedef enum uc_hook_type {
    // Hook all interrupt/syscall events
```

```
    UC_HOOK_INTR = 1 << 0,
    // Hook a particular instruction - only a very small subset of instructions
    // supported here
    UC_HOOK_INSN = 1 << 1,
    // Hook a range of code
    UC_HOOK_CODE = 1 << 2,
    // Hook basic blocks
    UC_HOOK_BLOCK = 1 << 3,
    // Hook for memory read on unmapped memory
    UC_HOOK_MEM_READ_UNMAPPED = 1 << 4,
    // Hook for invalid memory write events
    UC_HOOK_MEM_WRITE_UNMAPPED = 1 << 5,
    // Hook for invalid memory fetch for execution events
    UC_HOOK_MEM_FETCH_UNMAPPED = 1 << 6,
    // Hook for memory read on read-protected memory
    UC_HOOK_MEM_READ_PROT = 1 << 7,
    // Hook for memory write on write-protected memory
    UC_HOOK_MEM_WRITE_PROT = 1 << 8,
    // Hook for memory fetch on non-executable memory
    UC_HOOK_MEM_FETCH_PROT = 1 << 9,
    // Hook memory read events.
    UC_HOOK_MEM_READ = 1 << 10,
    // Hook memory write events.
    UC_HOOK_MEM_WRITE = 1 << 11,
    // Hook memory fetch for execution events
    UC_HOOK_MEM_FETCH = 1 << 12,
    // Hook memory read events, but only successful access.
    // The callback will be triggered after successful read.
    UC_HOOK_MEM_READ_AFTER = 1 << 13,
    // Hook invalid instructions exceptions.
    UC_HOOK_INSN_INVALID = 1 << 14,
    // Hook on new edge generation. Could be useful in program analysis.
    //
    // NOTE: This is different from UC_HOOK_BLOCK in 2 ways:
    //       1. The hook is called before executing code.
    //       2. The hook is only called when generation is triggered.
    UC_HOOK_EDGE_GENERATED = 1 << 15,
    // Hook on specific tcg op code. The usage of this hook is similar to
    // UC_HOOK_INSN.
    UC_HOOK_TCG_OPCODE = 1 << 16,
} uc_hook_type;
```

hook代码

Unicorn开始运行后，就可以通过代码的hook，查看具体的要执行的指令情况。

举例

官网代码

比如 官网测试代码 中的：

```
# callback for tracing instructions
def hook_code(uc, address, size, user_data):
    print(">>> Tracing instruction at 0x%x, instruction size = 0x%x" %(address, size))

...

        # tracing one instruction at ADDRESS with customized callback
        mu.hook_add(UC_HOOK_CODE, hook_code, begin=ADDRESS, end=ADDRESS)
```

其中的，此处的：

- hook代码的钩子参数叫：`UC_HOOK_CODE`
- hook代码的函数名：默认的，典型的，都叫做：`hook_code`
- hook_code内部的逻辑：此处只是简单的，打印出
  - address: 当前PC的地址 -> 表示代码执行到哪里了
  - size: 当前指令的字节大小

而此处只是用于演示，所以没有加更多复杂的逻辑。

下面介绍更加实例的，复杂的例子：

自己实例

以 模拟akd函数symbol2575 为例，其中的代码的hook部分是：

```
import re
from unicorn import *
from unicorn.arm64_const import *
from unicorn.arm_const import *

# only for debug
gNoUse = 0

#------------------- Code -------------------

# memory address where emulation starts
CODE_ADDRESS = 0x10000
logging.info("CODE_ADDRESS=0x%X", CODE_ADDRESS)
```

```
# code size: 4MB
CODE_SIZE = 4 * 1024 * 1024
logging.info("CODE_SIZE=0x%X", CODE_SIZE)
CODE_ADDRESS_END = (CODE_ADDRESS + CODE_SIZE) # 0x00410000
logging.info("CODE_ADDRESS_END=0x%X", CODE_ADDRESS_END)

CODE_ADDRESS_REAL_END = CODE_ADDRESS + gCodeSizeReal
logging.info("CODE_ADDRESS_REAL_END=0x%X", CODE_ADDRESS_REAL_END)
# CODE_ADDRESS_REAL_LAST_LINE = CODE_ADDRESS_REAL_END - 4
# logging.info("CODE_ADDRESS_REAL_LAST_LINE=0x%X", CODE_ADDRESS_REAL_LAST_LINE)

...

# callback for tracing instructions
def hook_code(mu, address, size, user_data):
...

    if pc == 0x12138:
        spValue = mu.mem_read(sp)
        logging.info("\tspValue=0x%X", spValue)
        gNoUse = 1

    if pc == 0x1213C:
        gNoUse = 1

    if pc == 0x118B4:
        gNoUse = 1

    if pc == 0x118B8:
        gNoUse = 1

...
        # tracing one instruction with customized callback
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS)
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_REA
L_END)
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=EMULATE_MALLOC_C
ODE_END)
        mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_END)
```

代码看起来，很多，很复杂。我们拆开来一点点解释：

添加代码hook

添加代码的hook，和官网示例中类似，都是：

```
mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_END)
```

其中：

- `UC_HOOK_CODE`：是要添加的hook类型，此处表示要去hook的是code代码
- `hook_code`：当执行到代码时，去调用到的（钩子）函数，即我们此处用于调试代码的函数
- `begin=CODE_ADDRESS, end=CODE_ADDRESS_END`：表示触发hook的代码的范围，此处值得说一下这个

细节：

- `begin=CODE_ADDRESS`：表示对于最初的代码的起始地址，这个没啥特殊的
- `end=CODE_ADDRESS_END`：这个值得重点解释一下
  - 第一层优化：官网示例代码的优化
    - 对于官网示例，默认是：`end=ADDRESS` ==代码起始地址 ->其最终的效果是：只触发了第一行的代码，其余代码都没触发此处的函数 `hook_code`
    - 所以我们此处要优化改进为：把范围放大到，真正的我们代码的范围
    - 所以之前用了代码：`mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_REAL_END)`，表示末尾处用的是 `end=CODE_ADDRESS_REAL_END`
  - 第二层优化：包含其他非当前函数的代码
    - 不过此处真正的代码的范围，按理说应该是 `end=CODE_ADDRESS_REAL_END`，但是此处没有采用，原因是：
      - 先贴出相关值

```
gCodeSizeReal=9416 == 0x24C8
CODE_ADDRESS=0x10000
CODE_SIZE=0x400000
CODE_ADDRESS_END=0x410000
CODE_ADDRESS_REAL_END=0x124C8
```

    - 如果用 `end=CODE_ADDRESS_REAL_END == 0x124C8`，则此处，的确是对于当前要模拟的函数的代码，是正常hook，都可以触发到 `hook_code` 了，但是，后续对于，
    - 除了 `CODE_ADDRESS ~ CODE_ADDRESS_REAL_END == 0x10000 ~ 0x124C8` 之外，
    - 在 `CODE_ADDRESS ~ CODE_ADDRESS_END == 0x10000 ~ 0x410000` 之内，
    - 还有些额外的代码，是此处特殊的，要模拟的第三方函数，比如 `malloc` 等，也想要触发调试函数 `hook_code`
    - 如果设置了 `end=CODE_ADDRESS_REAL_END`，则函数本身之外的其他额外代码，就无法触发 `hook_code`，导致无法调试内部细节了

其他细节

`hook_code` 中其他部分的代码，也很多，分别实现了各自的目的和效果，包括：

- 日志优化：打印当前正在执行的指令
  - 详见：优化日志输出
- 借助Capstone查看当前真正执行的指令
  - Capstone

其他一些细节，还有：

- 当特定PC时，查看读取内存的值

```
if pc == 0x12138:
    spValue = mu.mem_read(sp)
    logging.info("\tspValue=0x%X", spValue)
```

- 当特定PC时，暂停运行，用于辅助调试，查看其他的值

```
if pc == 0x1213C:
    gNoUse = 1
```

hook指令

Unicorn对于指令的hook的核心用法：

- 调用 `hook_add` 时传递参数
  - `type = UC_HOOK_INSN`
  - `ins = UC_{arch}_INS_{xxx}`
    - `arch`：架构，`X86 / ARM64` 等
    - `xxx`：指令名称
- 具体支持的指令
  - `X86`：支持很多指令的hook
    - 比如
      - `UC_X86_INS_SYSCALL`
      - `UC_X86_INS_IN`
      - `UC_X86_INS_OUT`
    - 其他指令详见：x86_const
  - `ARM64`：只支持极其有限的几个指令，其他指令不支持
    - `UC_ARM64_INS_MRS`
    - `UC_ARM64_INS_MSR`
    - `UC_ARM64_INS_SYS`
    - `UC_ARM64_INS_SYSL`

如何使用hook_add的UC_HOOK_INSN

官网有例子sample_arm64，供参考：

```
def test_arm64_hook_mrs():
  def _hook_mrs(uc, reg, cp_reg, _):
    print(f">>> Hook MRS instruction: reg = 0x{reg:x}(UC_ARM64_REG_X2) cp_reg = {cp_reg}
")
    uc.reg_write(reg, 0x114514)
    print(">>> Write 0x114514 to X")

    # Skip MRS instruction
    return True


. . .
    # Hook MRS instruction
    mu.hook_add(UC_HOOK_INSN, _hook_mrs, None, 1, 0, UC_ARM64_INS_MRS)
```

hook内存

在Unicorn模拟代码期间，常会涉及到调试查看内存的值，此时就涉及到：内存的hook

Unicorn对于内存的操作的hook，支持：

- `UC_HOOK_MEM_READ`
- `UC_HOOK_MEM_WRITE`

常见的用法有：

以 模拟akd函数symbol2575 为例，其中相关代码是：

```
def hook_mem_write(uc, access, address, size, value, user_data):
    pc = uc.reg_read(UC_ARM64_REG_PC)
    print(">>> Memory WRITE at 0x%X, size= %u, value= 0x%X, PC= 0x%X" % (address, size,
 value, pc))
    gNoUse = 1

def hook_mem_read(uc, access, address, size, value, user_data):
    pc = uc.reg_read(UC_ARM64_REG_PC)
    data = uc.mem_read(address, size)
    print("<<< Memory READ at 0x%X, size= %u, value= 0x%s, pc= 0x%X" % (address, size,
data.hex(), pc))
    gNoUse = 1

...
        # hook memory read and write
        mu.hook_add(UC_HOOK_MEM_READ, hook_mem_read)
        mu.hook_add(UC_HOOK_MEM_WRITE, hook_mem_write)
```

输出效果：

```
# [0x1001c] size=0x4, opcode=FD 7B 0D A9
  [0x1001c]     stp     x29, x30, [sp, #0xd0]
>>> Memory WRITE at 0x7FFFF0, size= 8, value= 0x0, PC= 0x1001C
>>> Memory WRITE at 0x7FFFF8, size= 8, value= 0x300000, PC= 0x1001C
. . .
# [0x10028] size=0x4, opcode=68 8F 2F 58
  [0x10028]     ldr     x8, #0x6f214
<<< Memory READ at 0x6F214, size= 8, value= 0x0000000000000000, pc= 0x10028
```

额外解释：

对于其中的`stp`指令触发了多个（2个）的Memeory Write，背后的逻辑是：

`stp`是ARM汇编指令，其中p是pair，一次性操作2个地址，即先后把2个值，分别写入对应的地址，所以才会触发2次的`UC_HOOK_MEM_WRITE`，输出2个log。

hook异常

此处的hook异常，主要指的是，当发生一些异常情况时的hook

其中对于内存memory来说，常见的异常就是：未映射（但却去读取或写入对应的）内存地址

对应的hook的类型是：

- `UC_HOOK_MEM_READ_UNMAPPED`
  - 内存地址未映射，但却去：读取内存
- `UC_HOOK_MEM_WRITE_UNMAPPED`
  - 内存地址未映射，但却去：写入内存
- `UC_HOOK_MEM_FETCH_UNMAPPED`
  - 内存地址未映射，但却去：内存取指
    - 取指=读取指令=（Unicorn模拟CPU去）从对应内存地址，读取指令（供后续模拟解析和运行）

以及其他方面的一些异常还有：

- `UC_HOOK_INSN_INVALID`
  - 非法指令

组合值

而其中具体使用的时候，更常见的用法是：

- 用组合出来的值
  - `UC_HOOK_MEM_UNMAPPED`
    - = `UC_HOOK_MEM_READ_UNMAPPED` + `UC_HOOK_MEM_WRITE_UNMAPPED` + `UC_HOOK_MEM_FETCH_UNMAPPED`
  - `UC_HOOK_MEM_PROT`
    - = `UC_HOOK_MEM_READ_PROT` + `UC_HOOK_MEM_WRITE_PROT` + `UC_HOOK_MEM_FETCH_PROT`
  - `UC_HOOK_MEM_READ_INVALID`
    - = `UC_HOOK_MEM_READ_PROT` + `UC_HOOK_MEM_READ_UNMAPPED`
  - `UC_HOOK_MEM_WRITE_INVALID`
    - = `UC_HOOK_MEM_WRITE_PROT` + `UC_HOOK_MEM_WRITE_UNMAPPED`
  - `UC_HOOK_MEM_FETCH_INVALID`
    - = `UC_HOOK_MEM_FETCH_PROT` + `UC_HOOK_MEM_FETCH_UNMAPPED`
  - `UC_HOOK_MEM_INVALID`
    - = `UC_HOOK_MEM_UNMAPPED` + `UC_HOOK_MEM_PROT`
  - `UC_HOOK_MEM_VALID`
    - = `UC_HOOK_MEM_READ` + `UC_HOOK_MEM_WRITE` + `UC_HOOK_MEM_FETCH`

对应的组合的值，官网源码中可以找到定义unicorn.h：

```
// Hook type for all events of unmapped memory access
#define UC_HOOK_MEM_UNMAPPED                                    \
```

```
    (UC_HOOK_MEM_READ_UNMAPPED + UC_HOOK_MEM_WRITE_UNMAPPED +                  \
     UC_HOOK_MEM_FETCH_UNMAPPED)
// Hook type for all events of illegal protected memory access
#define UC_HOOK_MEM_PROT                                                       \
    (UC_HOOK_MEM_READ_PROT + UC_HOOK_MEM_WRITE_PROT + UC_HOOK_MEM_FETCH_PROT)
// Hook type for all events of illegal read memory access
#define UC_HOOK_MEM_READ_INVALID                                               \
    (UC_HOOK_MEM_READ_PROT + UC_HOOK_MEM_READ_UNMAPPED)
// Hook type for all events of illegal write memory access
#define UC_HOOK_MEM_WRITE_INVALID                                              \
    (UC_HOOK_MEM_WRITE_PROT + UC_HOOK_MEM_WRITE_UNMAPPED)
// Hook type for all events of illegal fetch memory access
#define UC_HOOK_MEM_FETCH_INVALID                                              \
    (UC_HOOK_MEM_FETCH_PROT + UC_HOOK_MEM_FETCH_UNMAPPED)
// Hook type for all events of illegal memory access
#define UC_HOOK_MEM_INVALID (UC_HOOK_MEM_UNMAPPED + UC_HOOK_MEM_PROT)
// Hook type for all events of valid memory access
// NOTE: UC_HOOK_MEM_READ is triggered before UC_HOOK_MEM_READ_PROT and
// UC_HOOK_MEM_READ_UNMAPPED, so
//       this hook may technically trigger on some invalid reads.
#define UC_HOOK_MEM_VALID                                                      \
    (UC_HOOK_MEM_READ + UC_HOOK_MEM_WRITE + UC_HOOK_MEM_FETCH)
```

用法举例

以 模拟akd函数symbol2575 为例，其中的：

```
def hook_unmapped(mu, access, address, size, value, context):
    pc = mu.reg_read(UC_ARM64_REG_PC)
    logging.info("!!! Memory UNMAPPED at 0x%X size=0x%x, access(r/w)=%d, value=0x%X, PC
=0x%X", address, size, access, value, pc)
    mu.emu_stop()
    return True

def hook_mem_write(uc, access, address, size, value, user_data):
'''
    pc = uc.reg_read(UC_ARM64_REG_PC)
    logging.info(" >> Memory WRITE at 0x%X, size=%u, value=0x%X, PC=0x%X", address, size
, value, pc)
    # logging.info(" >> Memory WRITE at 0x%X, size=%u, value=0x%s, PC=0x%X", address, s
ize, value.to_bytes(8, "little").hex(), pc))
    gNoUse = 1

def hook_mem_read(uc, access, address, size, value, user_data):
    if address == ARG_routingInfoPtr:
        logging.info("read ARG_routingInfoPtr")
        gNoUse = 1

    pc = uc.reg_read(UC_ARM64_REG_PC)
    data = uc.mem_read(address, size)
    logging.info(" << Memory READ at 0x%X, size=%u, rawValueLittleEndian=0x%s, pc=0x%X",
 address, size, data.hex(), pc)
    gNoUse = 1
```

```
    dataLong = int.from_bytes(data, "little", signed=False)
    if dataLong == 0:
        logging.info(" !! Memory read out 0 -> possbile abnormal -> need attention")
        gNoUse = 1

# def hook_mem_fetch(uc, access, address, size, value, user_data):
#     pc = uc.reg_read(UC_ARM64_REG_PC)
#     logging.info(" >> Memory FETCH at 0x%X, size= %u, value= 0x%X, PC= 0x%X", address
, size, value, pc))
#     gNoUse = 1

...
        # hook unmamapped memory
        mu.hook_add(UC_HOOK_MEM_UNMAPPED, hook_unmapped)

        # hook memory read and write
        mu.hook_add(UC_HOOK_MEM_READ, hook_mem_read)
        mu.hook_add(UC_HOOK_MEM_WRITE, hook_mem_write)
        # mu.hook_add(UC_HOOK_MEM_FETCH, hook_mem_fetch)
```

就是对应的：

- `UC_HOOK_MEM_READ`：读取内存
- `UC_HOOK_MEM_WRITE`：写入内存
- `UC_HOOK_MEM_UNMAPPED`：内存发生未映射的错误异常

的hook的用法。

其中 `hook_mem_read` 中的：

```
    dataLong = int.from_bytes(data, "little", signed=False)
    if dataLong == 0:
        logging.info(" !! Memory read out 0 -> possbile abnormal -> need attention")
        gNoUse = 1
```

的含义是：代码模拟期间，发生很多次：

- 内存读取出来的值是0

具体的log类似于：

```
=== 0x000113E4 +5092 : E9 6F 40 B9  -> ldr     w9, [sp, #0x6c]
 << Memory READ at 0x77FF7C, size=4, rawValueLittleEndian=0x00000000, pc=0x113E4
```

而，从某个内存地址读取出来的值是0，往往又是：后续其他代码逻辑报错的原因

因为正常情况下，从内存读取出来的值，往往都不是0

所以，由此加了个log日志，变成：

```
=== 0x000113E4 +5092 : E9 6F 40 B9  -> ldr     w9, [sp, #0x6c]
 << Memory READ at 0x77FF7C, size=4, rawValueLittleEndian=0x00000000, pc=0x113E4
```

```
Memory read out 0 -> possbile abnormal -> need attention
```

用于提示，此时需要注意，方便后续调试时，找到最近的一处的，可能出错的地方 = 当内存读取值是0的地方。

打印日志

Unicorn模拟调试代码逻辑的过程中，少不了通过打印日志，实现调试变量值的目的。

而此处常用的Python代码中的打印日志，对应函数主要是：`print`

而如果要把日志输出到文件，则可以用`logging`库。

下面介绍一些，关于打印日志方面的心得。

优化日志输出

日志优化：通用且统一的方式打印寄存器值

自己的实际代码 模拟akd函数symbol2575 中的 `hook_code` 中的这部分代码：

```
# callback for tracing instructions
def hook_code(mu, address, size, user_data):
    global ucHeap

    pc = mu.reg_read(UC_ARM64_REG_PC)
...
    # for debug
    toLogDict = {
        0x00010070: ["x25"],
        0x00010074: ["cpsr", "w9", "x9", "x25"],
        0x00010078: ["cpsr", "x9"],
...
        0x00012450: ["x27"],
    }

    # common debug

    cpsr = mu.reg_read(UC_ARM_REG_CPSR)
    sp = mu.reg_read(UC_ARM_REG_SP)

    w8 = mu.reg_read(UC_ARM64_REG_W8)
    w9 = mu.reg_read(UC_ARM64_REG_W9)
    w10 = mu.reg_read(UC_ARM64_REG_W10)
    w11 = mu.reg_read(UC_ARM64_REG_W11)
    w24 = mu.reg_read(UC_ARM64_REG_W24)
    w26 = mu.reg_read(UC_ARM64_REG_W26)

    x0 = mu.reg_read(UC_ARM64_REG_X0)
    x1 = mu.reg_read(UC_ARM64_REG_X1)
    x2 = mu.reg_read(UC_ARM64_REG_X2)
    x3 = mu.reg_read(UC_ARM64_REG_X3)
    x4 = mu.reg_read(UC_ARM64_REG_X4)
    x8 = mu.reg_read(UC_ARM64_REG_X8)
    x9 = mu.reg_read(UC_ARM64_REG_X9)
    x10 = mu.reg_read(UC_ARM64_REG_X10)
    x16 = mu.reg_read(UC_ARM64_REG_X16)
    x22 = mu.reg_read(UC_ARM64_REG_X22)
    x24 = mu.reg_read(UC_ARM64_REG_X24)
    x25 = mu.reg_read(UC_ARM64_REG_X25)
    x26 = mu.reg_read(UC_ARM64_REG_X26)
    x27 = mu.reg_read(UC_ARM64_REG_X27)

    regNameToValueDict = {
        "cpsr": cpsr,
        "sp": sp,
```

```
        "w8": w8,
        "w9": w9,
        "w10": w10,
        "w11": w11,
        "w24": w24,
        "w26": w26,

        "x0": x0,
        "x1": x1,
        "x2": x2,
        "x3": x3,
        "x4": x4,
        "x8": x8,
        "x9": x9,
        "x10": x10,
        "x16": x16,
        "x22": x22,
        "x24": x24,
        "x25": x25,
        "x26": x26,
        "x27": x27,
    }

    toLogAddressList = toLogDict.keys()
    if pc in toLogAddressList:
        toLogRegList = toLogDict[pc]
        initLogStr = "\tdebug: PC=0x%X: " % pc
        regLogStrList = []
        for eachRegName in toLogRegList:
            eachReg = regNameToValueDict[eachRegName]
            isWordReg = re.match("x\d+", eachRegName)
            logFormt = "0x%016X" if isWordReg else "0x%08X"
            curRegValueStr = logFormt % eachReg
            curRegLogStr = "%s=%s" % (eachRegName, curRegValueStr)
            regLogStrList.append(curRegLogStr)
        allRegStr = ", ".join(regLogStrList)
        wholeLogStr = initLogStr + allRegStr
        logging.info("%s", wholeLogStr)
        gNoUse = 1
```

是优化后的，为了实现调试的目的：

希望调试当某个PC值时，去打印对应的寄存器的值

而之前都是，单个的PC地址，分别写调试代码，效率很低。

所以最后统一成此处的代码：

通用的输出log，打印寄存器的代码

而想要新增一个调试时，只需要单独给 `toLogDict` 加一行定义，比如：

- `0x00010074: ["cpsr", "w9", "x9", "x25"],`

就可以实现：

- 当PC值是 `0x00010074` 时，打印这些寄存器的值：cpsr、w9、x9、x25

即可输出类似效果：

```
=== 0x00010074   +116 : 29 DB A9 B8   -> ldrsw   x9, [x25, w9, sxtw #2]
    debug: PC 0x10074: cpsr 0x20000000, w9 0x00000008, x9 0x0000000000000008, x25 0x000
0000000032850
```

实现我们的调试目的：查看此时特定寄存器的值，是否符合我们的预期。

注：后续如果要打印其他此处未定义的寄存器（比如 `x6` 等等），自己单独添加定义： `x6 = mu.reg_read(UC_ARM64_REG_X6)` 和 `regNameToValueDict` 中加上 `"x6": x6,` 即可。

查看当前指令

Unicorn模拟CPU去执行（函数）代码指令时，由于调试需要，往往需要查看（搞清楚）当前正在执行的指令是什么。

而关于Unicorn当前正在执行的指令是什么：

- Unicorn内部肯定是有的：在涉及到二进制代码解析时
  - 但是只有指令解析的结果，而具体的指令是什么，则：没有提供外部的任何接口
    - 所以Unicorn中，无法获取到，对应的是什么指令，这方面的信息
- 如果想要搞清楚：当前是什么指令
  - 暂时只能：引入外部的反汇编器disassembler，比如 `Capstone` ，自己去把二进制翻译为对应指令
    - 注：而 `Capstone` 翻译的指令，和 `Unicorn` 底层实际上所运行的指令：
      - 一般来说应该是一样的
        - 当然，大部分时候，也的确是一样的
      - 但按理说，也可以是不一样的
        - 比如某些极其特殊的情况
          - 举例
            - `Unicorn` 内部用到的代码解析是，比如说是只支持 `ARM64` 的，而额外引用的 `Casptone` ，比如说支持新的 `arm64e` 架构
              - 可能会出现，对于同样的二进制 `7F 23 03 D5` ， `Unicorn` 内部被翻译为 `HINT` 指令，而 `Capstone` （反汇编）打印出是（实际上arm64e中才支持的）PAC相关指令： `pacibsp`
    - 详见独立子教程：反汇编利器：Capstone

运行后

在Unicorn模拟代码运行之后，所涉及到的内容主要有：

- 如果不设置合适的停止时机，往往Unicorn会一直运行下去，所以找个合适时机去停止运行
  - 往往都是 `ret` 指令时，去调用 `emu_stop` 去停止
- 运行后，获取程序输出的结果

下面分别详细解释：

停止运行

对于Unicorn来说，就是模拟CPU运行，模拟去读取指令和运行指令而已。

所以，换句话说，如果你的给code代码的地址空间写入了代码后，如果没有额外的跳转等复杂逻辑，则：

- **Unicorn会一直运行下去**

如果没有合适的触发时机，去让其停下来，那就变成了死循环，永远不结束了。

而我们的目标是：模拟代码，尤其是函数的逻辑，希望代码运行完毕，输出结果的。

所以，此处往往选择一个合适的时机去触发其让Unicorn停下来。

这个时机，一般都是：`ret`指令，即，当发现正在运行的指令是`ret`指令，则就会调用`emu_stop`去停下来。

举例

自己的实例

以 模拟akd函数symbol2575 为例，其中的Stop的判断逻辑是：

```
import re

#------------------- Code -------------------

# memory address where emulation starts
CODE_ADDRESS = 0x10000

# code size: 4MB
CODE_SIZE = 4 * 1024 * 1024
CODE_ADDRESS_END = (CODE_ADDRESS + CODE_SIZE) # 0x00410000

CODE_ADDRESS_REAL_END = CODE_ADDRESS + gCodeSizeReal

def shouldStopEmulate(curPc, decodedInsn):
    isShouldStop = False
    # isRetInsn = decodedInsn.mnemonic == "ret"
    isRetInsn = re.match("^ret", decodedInsn.mnemonic) # support: ret/retaa/retab/...
    if isRetInsn:
        isPcInsideMainCode = (curPc >= CODE_ADDRESS) and (curPc < CODE_ADDRESS_REAL_END)

        isShouldStop = isRetInsn and isPcInsideMainCode

    return isShouldStop

# callback for tracing instructions
def hook_code(mu, address, size, user_data):
...
    lineCount = int(size / BYTES_PER_LINE)
    for curLineIdx in range(lineCount):
```

```
...
        decodedInsnGenerator = cs.disasm(opcodeBytes, address)
        for eachDecodedInsn in decodedInsnGenerator:
            eachInstructionName = eachDecodedInsn.mnemonic

            if shouldStopEmulate(pc, eachDecodedInsn):
                mu.emu_stop()
                logging.info("Emulate done!")

            gNoUse = 1

...

        mu.hook_add(UC_HOOK_CODE, hook_code, begin CODE_ADDRESS, end CODE_ADDRESS_END)
```

其中的主要逻辑是：

在 `hook_code` 中，借助 `Capstone` 反编译出当前指令，其中 `mnemonic` 就是指令名称，当发现是 `ret` 指令时

注：对于arm64e来说，还有更多的PAC相关ret指令：`retaa` 、`retab` 等，所以此处用 `re` 正则去判断指令名称是否匹配，而不是直接判断和 `ret` 是否相等。

就去调用 `emu_stop()` 去停止Unicorn的继续运行。

此处有个细节：

此处判断代码结束的条件是：`isShouldStop = isRetInsn and isPcInsideMainCode`

除了：

- `isRetInsn` ：判断代码是否是ret指令

还有个：

- `isPcInsideMainCode` ：判断代码是否在主体的main的，要模拟运行的函数代码内部，才返回
  - 目的是：防止，触发了别处的特殊代码中的ret，也返回了。
    - 比如别处特殊的代码，就包括，后续的 调用其他子函数 中，在模拟malloc等函数时，其内部也是有 `ret` 指令的，此时只是子函数的返回，而不应该是，整个Unicorn的stop
  - 所以要排除这类特殊情况，只是当代码地址在main函数=要模拟的函数内部时，其ret才是真正要返回，要Unicorn停止的意思。

获取结果

只有当，实现了前面的

Unicorn停止运行

后，在`emu_start`的之后的代码，才会运行到。

然后这部分代码，也往往就是：去获取程序（函数）运行的结果，得到最终的返回值。

举例

自己的实例

以 模拟akd函数symbol2575 为例，就是：

```
        # emulate machine code in infinite time
        mu.emu_start(CODE_ADDRESS, CODE_ADDRESS + len(ARM64_CODE_akd_symbol2575))

        # now print out some registers
        logging.info("---------- Emulation done. Below is the CPU context ----------")

        retVal = mu.reg_read(UC_ARM64_REG_X0)
        # routingInfo = mu.mem_read(ARG_routingInfoPtr)
        # logging.info(">>> retVal=0x%x, routingInfo=%d", retVal, routingInfo))
        logging.info(">>> retVal=0x%x", retVal)

        routingInfoEnd = mu.mem_read(ARG_routingInfoPtr, 8)
        logging.info(">>> routingInfoEnd hex=0x%s", routingInfoEnd.hex())
        routingInfoEndLong = int.from_bytes(routingInfoEnd, "little", signed=False)
        logging.info(">>> routingInfoEndLong=%d", routingInfoEndLong)
```

中的：

```
        # now print out some registers
        logging.info("---------- Emulation done. Below is the CPU context ----------")

        retVal = mu.reg_read(UC_ARM64_REG_X0)
        # routingInfo = mu.mem_read(ARG_routingInfoPtr)
        # logging.info(">>> retVal=0x%x, routingInfo=%d", retVal, routingInfo))
        logging.info(">>> retVal=0x%x", retVal)

        routingInfoEnd = mu.mem_read(ARG_routingInfoPtr, 8)
        logging.info(">>> routingInfoEnd hex=0x%s", routingInfoEnd.hex())
        routingInfoEndLong = int.from_bytes(routingInfoEnd, "little", signed=False)
        logging.info(">>> routingInfoEndLong=%d", routingInfoEndLong)
```

也就是我们希望的，此处获取对应的返回值的代码逻辑。

而此处特定的要模拟的函数 `arm64` 的 `___lldb_unnamed_symbol2575$$akd` 函数的返回值，是通过代码 `mu.mem_read(ARG_routingInfoPtr, 8)` ，从传入的指针 `ARG_routingInfoPtr` 中获取返回值

而函数本身的返回值，则是普通的逻辑，通过代码 `mu.reg_read(UC_ARM64_REG_X0)` 去读取ARM中的寄存器 `x0` 的值

如此，即可获取到我们希望的返回的值了。

经验和心得

此处整理Unicorn开发期间的经验和心得：

无法跳过当前指令

- 需求：当发现是某些特定的指令时，想要跳过对应的指令，不去执行
- 结论：无法实现。Unicorn中不支持这类操作。
  - 具体详见：
    - 【未解决】unicorn模拟ARM汇编如何忽略特定指令为nop空指令

常见错误

此处整理Unicorn调试期间常遇到的错误以及背后的原因和对应的解决办法：

ERROR: Invalid memory mapping (UC_ERR_MAP)

- 现象：`ERROR: Invalid memory mapping (UC_ERR_MAP) == Memory UNMAPPED at`
- 原因：
  - 表面原因：多数是，内存地址是`0`或另外某个未映射的地址
  - 深层原因：模拟的代码前面的中间的某些步骤，已经出错了，导致此处的异常
    - 比如之前就读取某个地址的值是0
      - 其实是需要实现准备好环境：向对应地址写入对应的值，即可避免规避此问题

举例说明

- 【已解决】Unicorn模拟___lldb_unnamed_symbol2575$$akd：PC在+380处Invalid memory mapping UC_ERR_MAP

中的：

```
=== 0x0001016C  <+364> : 28 DB A8 B8  -> ldrsw   x8, [x25, w8, sxtw #2]
 << Memory READ at 0x32858, size=4, rawValueLittleEndian=0x00000000, pc=0x1016C
=== 0x00010170  <+368> : 1F 20 03 D5  -> nop
=== 0x00010174  <+372> : AA 5C 2C 58  -> ldr     x10, #0x68d08
 << Memory READ at 0x68D08, size=8, rawValueLittleEndian=0x0000000000000000, pc=0x10174
=== 0x00010178  <+376> : 08 01 0A 8B  -> add     x8, x8, x10
=== 0x0001017C  <+380> : 00 01 1F D6  -> br      x8
||| Memory UNMAPPED at 0x0 size=0x4, access(r/w)=21, value=0x0, PC=0x0
ERROR: Invalid memory mapping (UC_ERR_MAP)
```

需要对于上述，前面出现的`Memory READ`的`rawValueLittleEndian=0x0000000000000000`，即内存读取出来值是0的地方，写入特定的值

此特定的值，需要调试真实代码才能得到

此次调试后得到的值是：

```
<+380>: 00 01 1F D6 -> br x8
```

- 真实值
  - br x8
    - 要去跳转到：+484
      - +484
        - = x8 + x10
        - = 0x00000000000000c4 + 0x0000000102e70580
        - = 0x00000000000000c4 + akd`___lldb_unnamed_symbol2575$$akd + 288
        - = 0x0000000102e70644
        - = akd`___lldb_unnamed_symbol2575$$akd + 484

而：

- x10=akd`___lldb_unnamed_symbol2575$$akd + 288

此处模拟值： 有2个：

- 针对于
  - <+364>: 28 DB A8 B8 -> ldrsw x8, [x25, w8, sxtw #2]
- 要写入地址：x8 = 0x32858
  - 要写入值：0x00000000000000c4 和：
- 针对于
  - <+372>: AA 5C 2C 58 -> ldr x10, #0x68d08
- 要写入地址：x10 = 0x68D08
  - 要写入值：函数起始地址 + 288
    - = 0x10000 + 0x120
    - = 0x10120

所以解决办法是：

去写入对应的值：

```
  writeMemory(0x32858, 0xc4, 8)           # <+364>: 28 DB A8 B8  -> ldrsw   x8, [x25, w
8, sxtw #2]
  writeMemory(0x68D08, 0x10120, 8)        # <+372>: AA 5C 2C 58  -> ldr     x10, #0x68d
08

  0x0001016C: ["w8", "x25"],
  0x00010170: ["x8"],
  0x00010178: ["x10"],
```

然后解决解决问题，输出log中可以看出

```
=== 0x0001016C  <+364> : 28 DB A8 B8  -> ldrsw   x8, [x25, w8, sxtw #2]
    debug: PC 0x1016C: w8 0x00000002, x25 0x0000000000032850
 << Memory READ at 0x32858, size 4, rawValueLittleEndian 0xc4000000, pc 0x1016C
=== 0x00010170  <+368> : 1F 20 03 D5  -> nop
    debug: PC 0x10170: x8 0x00000000000000C4
=== 0x00010174  <+372> : AA 5C 2C 58  -> ldr     x10, #0x68d08
 << Memory READ at 0x68D08, size 8, rawValueLittleEndian 0x2001010000000000, pc 0x10174
=== 0x00010178  <+376> : 08 01 0A 8B  -> add     x8, x8, x10
    debug: PC 0x10178: x10 0x0000000000010120
=== 0x0001017C  <+380> : 00 01 1F D6  -> br      x8
>>> Tracing basic block at 0x101e4, block size = 0x48
=== 0x000101E4  <+484> : 16 00 80 52  -> movz    w22, #0
=== 0x000101E8  <+488> : 08 F3 8B 52  -> movz    w8, #0x5f98
```

代码在 `<+380>` 之后，可以正常继续运行了。

批量解决UC_ERR_MAP

后来发现此处被调试的程序，有个通用的逻辑：

给特定内存地址写入特定值，就可以批量的、一次性的，解决后续的br跳转报错UC_ERR_MAP的问题了

注：按照最新的理解，应该也算是，部分的实现了 `代码反混淆` = `反代码混淆` = `代码去混淆` 了。

- 概述：通过批量写入x9、x10地址相关值，确保了一次性解决了br跳转的问题
  - 详见
    - 【已解决】Unicorn模拟arm64代码：尝试批量一次性解决全部的br跳转导致内存映射错误UC_ERR_MAP
- 详解

搞懂___lldb_unnamed_symbol2575$$akd函数br跳转涉及到的固定的值的内存范围

- 【已解决】Unicorn模拟arm64代码：搞懂___lldb_unnamed_symbol2575$$akd函数br跳转涉及到的固定的值的内存范围
- 【已解决】Unicorn模拟arm64代码：计算br跳转涉及到的x10函数偏移量地址的合适的范围
- 【已解决】Unicorn模拟arm64代码：计算br跳转涉及到的x9的合适的范围

通过调试研究：

（1）计算br跳转涉及到的x9的合适的范围 是：

- 0x0000000100d91680 ~ 0x0000000100d938b0
  - = 函数起始地址(0x0000000100d70460)加上：0x21220 ~ 0x23450

对应的内存中保存的数据是：

```
(lldb) x 16gw 0x0000000100d91680
0x100d91680: 0x00000020 0x000000c0 0x0000005c 0x00000044
0x100d91690: 0x00000068 0x00000148 0x00000040 0x00000084
0x100d916a0: 0x00001ca0 0x00000028 0x00000068 0x000001fc
0x100d916b0: 0x00000000 0x0000008c 0x00000030 0x00000068

(lldb) x 16gw 0x0000000100d93880
0x100d93880: 0x00000020 0x00000044 0x0000006c 0x000005c4
0x100d93890: 0x00000994 0x000012e0 0x0000005c 0x000000a0
0x100d938a0: 0x00000000 0x000002d8 0x00000068 0x0000043c
0x100d938b0: 0x1001ffff 0x0002b400 0x0c02b400 0xc0000388
```

-> 0x100d91680之后，0x100d938b0之前，都是：小数 像是 偏移量地址。

（2）计算br跳转涉及到的x10函数偏移量地址的合适的范围 是

- x10：保存当前函数有效地址（0x0000000100d6xxx 或 0x0000000100d7xxx）的地址范围是：
  - 0x100dc8480 - 0x100dc9fe0

对应的内存中保存的数据是：

```
(lldb) x 16gx 0x100dc8440
0x100dc8440: 0x0000000000000200 0x0000000000000000
0x100dc8450: 0x0000000000000030 0x0000000100ce6d98
0x100dc8460: 0x0000000100cd4ec8 0x0000000100dab23b
0x100dc8470: 0x0000000000000200 0x0000000000000000
0x100dc8480: 0x0000000100d646a8 0x0000000100d64710
0x100dc8490: 0x0000000100d6476c 0x0000000100d647f0
0x100dc84a0: 0x0000000100d64818 0x0000000100d64880
0x100dc84b0: 0x0000000100d64924 0x0000000100d64a34

(lldb) x 16gx 0x100dc9fA0
0x100dc9fa0: 0x0000000100d7d2b4 0x0000000100d7d2d8
0x100dc9fb0: 0x0000000100d7d364 0x0000000100d7d38c
0x100dc9fc0: 0x0000000100d7d4a8 0x0000000100d7d228
0x100dc9fd0: 0x0000000100d7d294 0x0000000100d7c0e4
0x100dc9fe0: 0x00000001a51b2fc7 0x0000000192108102
0x100dc9ff0: 0x0000000100d933b2 0x0000000100de926a
0x100dca000: 0x0000000100de92b0 0x00000001db0441de
0x100dca010: 0x0000000100de92bb 0x0000000100de92b6
```

以及计算偏移量：

```
(lldb) p/x 0x100dc8480 - 0x0000000100d70460
(long) $16 = 0x0000000000058020

(lldb) p/x 0x100dc9fe0 - 0x0000000100d70460
(long) $17 = 0x0000000000059b80
```

=> 最后基本上判断出：

涉及到 `___lldb_unnamed_symbol2575$$akd` 的 `br` 跳转的内存地址范围，主要有2个段：

- **x9**的：`0x100d91680 ~ 0x100d938b0`
  - 当前函数起始地址：`0x0000000100d70460`
    - = 函数起始地址的偏移量：`+0x21220 ~ +0x23450`
- **x10**的：`0x100dc8480 - 0x100dc9fe0`
  - 当前函数起始地址：`0x0000000100d70460`
    - = 函数起始地址的偏移量：`+0x58020 ~ +0x59b80`

导出lldb调试时x9和x10两个段的实际数据值到文件

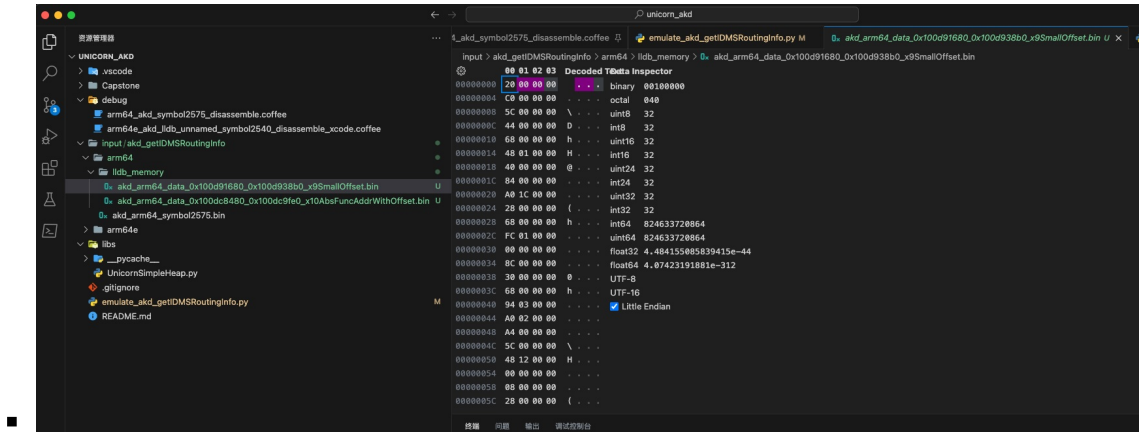
- 【已解决】Unicorn模拟arm64代码：导出lldb调试时x9和x10两个段的实际数据值到文件

通过调试实际代码期间，用lldb去导出的相关的x9和x10的数据文件：

- x9的
  - lldb导出命令：

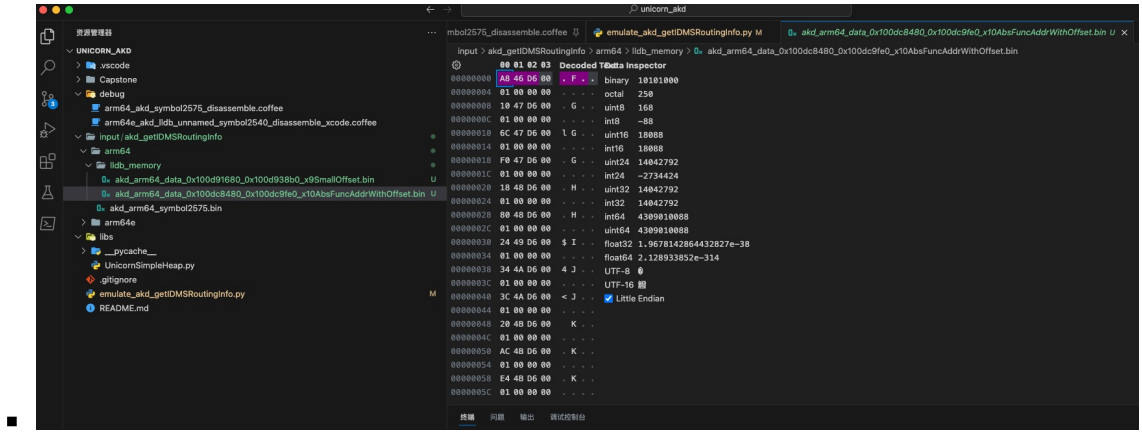
```
(lldb) memory read --binary --force --outfile /Users/crifan/dev/tmp/lldb_mem_du
mp/akd_arm64_data_0x100d91680_0x100d938b0_x9SmallOffset.bin 0x100d91680 0x100d9
```

```
38b0
8752 bytes written to '/Users/crifan/dev/tmp/lldb_mem_dump/akd_arm64_data_0x100d91680_0x100d938b0_x9SmallOffset.bin'
```

    - 导出的x9的文件
      - 

- x10的
  - lldb导出命令：

```
(lldb) memory read --binary --force --outfile /Users/crifan/dev/tmp/lldb_mem_dump/akd_arm64_data_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin 0x100dc8480 0x100dc9fe0
7008 bytes written to '/Users/crifan/dev/tmp/lldb_mem_dump/akd_arm64_data_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin'
```

  - 导出的x10的文件
    - 

修正导出的x10的带偏移量的函数地址

- 【已解决】Unicorn模拟arm64：修正导出的x10的带偏移量的函数地址

其中导出x10的数据中的地址，是absolute的绝对路径，所以此处还要去修复地址

即修改为此处Unicorn模拟时的，函数起始地址+相对偏移量=Unicorn模拟时的地址

用代码：

```
# Function: process akd_arm64_data_x10AbsFuncAddrWithOffset_fixOffset.bin offset
#   read out stored function abs address
```

```
#   remove function start address
#   add new Unicorn emulate address
#   write back
# Author: Crifan Li
# Update: 20230515

################################################################################
# Global Variable
################################################################################

gNoUse = 0

VALUE_GROUP_BYTE_LEN = 8
REAL_FUNC_START_ADDR = 0x0000000100d70460
EMULATE_FUNC_START_ADDR = 0x10000

################################################################################
# Util Function
################################################################################

def readBinFileBytes(inputFilePath):
    fileBytes = None
    with open(inputFilePath, "rb") as f:
        fileBytes = f.read()
    return fileBytes

def writeBytesToFile(outputFile, bytesData):
    with open(outputFile, "wb") as f:
        f.write(bytesData)

################################################################################
# Main
################################################################################

inputX10FuncAbsOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_d
ata_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin"
print("inputX10FuncAbsOffsetFile=%s" % inputX10FuncAbsOffsetFile)
inputX10AddrBytes = readBinFileBytes(inputX10FuncAbsOffsetFile) # b'\xa8F\xd6\x00\x01\x
00\x00\x00\x10G\xd6\x00\x01\x00\x00\x00lG\xd6\x00\x01 ...
inputX10AddrBytesLen = len(inputX10AddrBytes)
print("inputX10AddrBytesLen=%d == 0x%X" % (inputX10AddrBytesLen, inputX10AddrBytesLen))
inputX10AddrValueLen = (int)(inputX10AddrBytesLen / VALUE_GROUP_BYTE_LEN)
print("inputX10AddrValueLen=%d == 0x%X" % (inputX10AddrValueLen, inputX10AddrValueLen))

outputX10EmulateAddrValueBytes = bytearray()

for eachValueIdx in range(inputX10AddrValueLen):
    bytesStartIdx = eachValueIdx * VALUE_GROUP_BYTE_LEN
    bytesEndIdx = (bytesStartIdx + VALUE_GROUP_BYTE_LEN) - 1
    realAddrValueBytes = inputX10AddrBytes[bytesStartIdx:bytesEndIdx]
    realAddrValue = int.from_bytes(realAddrValueBytes, byteorder="little", signed=False)

    relativeOffset = realAddrValue - REAL_FUNC_START_ADDR
    relativeAbsOffset = abs(relativeOffset)
    emulateAddr = EMULATE_FUNC_START_ADDR + relativeOffset
    emuAddrBytes = int.to_bytes(emulateAddr, 8, byteorder="little", signed=False)
```

```
    outputX10EmulateAddrValueBytes.extend(emuAddrBytes)
    # print("outputX10EmulateAddrValueBytes=%s" % outputX10EmulateAddrValueBytes.hex())
    print("[0x%04X-0x%04X]=0x%s==0x%016X -> off:%d=abs(0x%X)->emu:0x%X>>%s" % (bytesSta
rtIdx, bytesEndIdx, realAddrValueBytes.hex(), realAddrValue, relativeOffset, relativeAb
sOffset, emulateAddr, emuAddrBytes.hex()))
    gNoUse = 0

outputX10EmulateAddrValueByteLen = len(outputX10EmulateAddrValueBytes)
print("\noutputX10EmulateAddrValueByteLen=%d=0x%X" % (outputX10EmulateAddrValueByteLen,
 outputX10EmulateAddrValueByteLen))

outputX10EmulateFuncAddrFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm6
4_data_x10EmulateAddr.bin"
print("outputX10EmulateFuncAddrFile=%s" % outputX10EmulateFuncAddrFile)

writeBytesToFile(outputX10EmulateFuncAddrFile, outputX10EmulateAddrValueBytes)

gNoUse = 0
```

输出文件:

```
input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_x10EmulateAddr.bin
```

输出log:

```
inputX10FuncAbsOffsetFile=input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data
_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin
inputX10AddrBytesLen=7008 == 0x1B60
inputX10AddrValueLen=876 == 0x36C
[0x0000-0x0007] 0xa846d600010000 -> 0x0000000100D646A8 -> off:-48568 abs(0xBDB8)- emu:0x4
248 -> 4842000000000000
[0x0008-0x000F] 0x1047d600010000 -> 0x0000000100D64710 -> off:-48464 abs(0xBD50)- emu:0x4
2B0 -> b042000000000000
[0x0010-0x0017] 0x6c47d600010000 -> 0x0000000100D6476C -> off:-48372 abs(0xBCF4)- emu:0x4
30C -> 0c43000000000000
[0x0018-0x001F] 0xf047d600010000 -> 0x0000000100D647F0 -> off:-48240 abs(0xBC70)- emu:0x4
390 -> 9043000000000000
[0x0020-0x0027] 0x1848d600010000 -> 0x0000000100D64818 -> off:-48200 abs(0xBC48)- emu:0x4
3B8 -> b843000000000000
[0x0028-0x002F] 0x8048d600010000 -> 0x0000000100D64880 -> off:-48096 abs(0xBBE0)- emu:0x4
420 -> 2044000000000000
[0x0030-0x0037] 0x2449d600010000 -> 0x0000000100D64924 -> off:-47932 abs(0xBB3C)- emu:0x4
4C4 -> c444000000000000
[0x0038-0x003F] 0x344ad600010000 -> 0x0000000100D64A34 -> off:-47660 abs(0xBA2C)- emu:0x4
5D4 -> d445000000000000
[0x0040-0x0047] 0x3c4ad600010000 -> 0x0000000100D64A3C -> off:-47652 abs(0xBA24)- emu:0x4
5DC -> dc45000000000000
[0x0048-0x004F] 0x204bd600010000 -> 0x0000000100D64B20 -> off:-47424 abs(0xB940)- emu:0x4
6C0 -> c046000000000000
[0x0050-0x0057] 0xac4bd600010000 -> 0x0000000100D64BAC -> off:-47284 abs(0xB8B4)- emu:0x4
74C -> 4c47000000000000
[0x0058-0x005F] 0xe44bd600010000 -> 0x0000000100D64BE4 -> off:-47228 abs(0xB87C)- emu:0x4
784 -> 8447000000000000

...
```

```
[0x1B28-0x1B2F] 0xd8d2d700010000 -> 0x0000000100D7D2D8 -> off:52856 abs(0xCE78)- emu:0x1CE78 -> 78ce010000000000
[0x1B30-0x1B37] 0x64d3d700010000 -> 0x0000000100D7D364 -> off:52996 abs(0xCF04)- emu:0x1CF04 -> 04cf010000000000
[0x1B38-0x1B3F] 0x8cd3d700010000 -> 0x0000000100D7D38C -> off:53036 abs(0xCF2C)- emu:0x1CF2C -> 2ccf010000000000
[0x1B40-0x1B47] 0xa8d4d700010000 -> 0x0000000100D7D4A8 -> off:53320 abs(0xD048)- emu:0x1D048 -> 48d0010000000000
[0x1B48-0x1B4F] 0x28d2d700010000 -> 0x0000000100D7D228 -> off:52680 abs(0xCDC8)- emu:0x1CDC8 -> c8cd010000000000
[0x1B50-0x1B57] 0x94d2d700010000 -> 0x0000000100D7D294 -> off:52788 abs(0xCE34)- emu:0x1CE34 -> 34ce010000000000
[0x1B58-0x1B5F] 0xe4c0d700010000 -> 0x0000000100D7C0E4 -> off:48260 abs(0xBC84)- emu:0x1BC84 -> 84bc010000000000

outputX10EmulateAddrValueByteLen=7008=0x1B60
outputX10EmulateFuncAddrFile=input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_x10EmulateAddr.bin
```

把导出的x9和x10的2段数据导入到Unicorn模拟代码中

- 【已解决】Unicorn模拟arm64代码：把导出的x9和x10的2段数据导入到Unicorn模拟代码中

而关于批量写入x9、x10地址相关值，用的代码是：

```
#-------------------- Try fix br jump UC_ERR_MAP --------------------

x9SmallOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_0x100d91680_0x100d938b0_x9SmallOffset.bin"
print("x9SmallOffsetFile=%s" % x9SmallOffsetFile)
x9SmallOffsetBytes = readBinFileBytes(x9SmallOffsetFile)
x9SmallOffsetBytesLen = len(x9SmallOffsetBytes) # b' \x00\x00\x00\xc0\x00\x00\x00\\\x00\x00\x00D\x00\x00\x00h\x00\x00\x00H\x01 ...
print("x9SmallOffsetBytesLen=%d=0x%X" % (x9SmallOffsetBytesLen, x9SmallOffsetBytesLen))

x9SmallOffsetStartAddress = CODE_ADDRESS + 0x21220
print("x9SmallOffsetStartAddress=0x%X" % x9SmallOffsetStartAddress)

# x10AbsFuncAddrWithOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin"
x10AbsFuncAddrWithOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_x10EmulateAddr.bin"
print("x10AbsFuncAddrWithOffsetFile=%s" % x10AbsFuncAddrWithOffsetFile)
x10AbsFuncAddrWithOffsetBytes = readBinFileBytes(x10AbsFuncAddrWithOffsetFile)
# x10AbsFuncAddrWithOffsetBytesLen = len(x10AbsFuncAddrWithOffsetBytes) # b'\xa8F\xd6\x00\x01\x00\x00\x00\x10G\xd6\x00\x01\x00\x00\x00lG\xd6\x00\x01 ...
x10AbsFuncAddrWithOffsetBytesLen = len(x10AbsFuncAddrWithOffsetBytes) # b'HB\x00\x00\x00\x00\x00\x00\x00\xb0B\x00\x00\x00\x00\x00\x00\x0cC\x00\x00\x00\ ...
print("x10AbsFuncAddrWithOffsetBytesLen=%d=0x%X" % (x10AbsFuncAddrWithOffsetBytesLen, x10AbsFuncAddrWithOffsetBytesLen)) # x10AbsFuncAddrWithOffsetBytesLen=7008=0x1B60

x10AbsFuncAddrWithOffsetStartAddress = CODE_ADDRESS + 0x58020
print("x10AbsFuncAddrWithOffsetStartAddress=0x%X" % x10AbsFuncAddrWithOffsetStartAddress
```

```
)

'''

        mu.mem_write(x9SmallOffsetStartAddress, x9SmallOffsetBytes)
        print(" >> has write %d=0x%X bytes into memory [0x%X-0x%X]" % (x9SmallOffsetByt
esLen, x9SmallOffsetBytesLen, x9SmallOffsetStartAddress, x9SmallOffsetStartAddress + x9
SmallOffsetBytesLen))
        mu.mem_write(x10AbsFuncAddrWithOffsetStartAddress, x10AbsFuncAddrWithOffsetBytes
)
        print(" >> has write %d=0x%X bytes into memory [0x%X-0x%X]" % (x10AbsFuncAddrWi
thOffsetBytesLen, x10AbsFuncAddrWithOffsetBytesLen, x10AbsFuncAddrWithOffsetStartAddress
, x10AbsFuncAddrWithOffsetStartAddress + x10AbsFuncAddrWithOffsetBytesLen))
'''
        # writeMemory(0x32850, 0x00000094, 4)             # <+236>: 29 DB A9 B8  -> ldr
sw   x9, [x25, w9, sxtw #2]
        # readMemory(0x32850, 4)
        # writeMemory(0x32870, 0xffffdbc4, 4)     # <+116>: 29 DB A9 B8  -> ldrsw   x9,
 [x25, w9, sxtw #2]
        # readMemory(0x32870, 4)
        # writeMemory(0x68CF8, CODE_ADDRESS_REAL_END, 8)  # <+124>: EA 63 2C 58  -> ldr
    x10, #0x68cf8
        # readMemory(0x68CF8, 8)
        # writeMemory(0x68D00, 0x1008C, 8)        # <+244>: 6A 60 2C 58  -> ldr     x10
, #0x68d00
        # readMemory(0x68D00, 8)
        # writeMemory(0x32858, 0xc4, 4)           # <+364>: 28 DB A8 B8  -> ldrsw   x8,
 [x25, w8, sxtw #2]
        # readMemory(0x32858, 4)
        # writeMemory(0x68D08, 0x10120, 8)        # <+372>: AA 5C 2C 58  -> ldr     x10
, #0x68d08
        # readMemory(0x68D08, 8)
```

即可，批量的，一次性的解决了之前br跳转导致内存映射报错的问题。

效果对比：

- 之前：用Unicorn调试，耗时很久，才一点点调试到300多行，且遇到多次br跳转问题
- 现在：这下代码至少一次性的运行到了 <+4404> 即 4000多行，才出现其他问题。

ERROR: Invalid memory write (UC_ERR_WRITE_UNMAPPED)

- 现象

代码：

```
# Stack: from High address to lower address ?
STACK_ADDRESS = 8 * 1024 * 1024
STACK_SIZE = 1 * 1024 * 1024
STACK_ADDRESS_END = STACK_ADDRESS - STACK_SIZE # 7 * 1024 * 1024

STACK_SP = STACK_ADDRESS - 0x8 # ARM64: offset 0x8
...
        # map stack
        mu.mem_map(STACK_ADDRESS, STACK_SIZE)
```

报错：`ERROR: Invalid memory write (UC_ERR_WRITE_UNMAPPED)`

- 原因：此处Stack堆栈初始化有问题：Stack的map时的起始地址，有误，写成了Stack的高地址了
- 解决办法：把Stack的起始地址改为，内存的低地址（而不是高地址）
- 具体做法：

代码改为：

```
        # mu.mem_map(STACK_ADDRESS, STACK_SIZE)
        mu.mem_map(STACK_ADDRESS_END, STACK_SIZE)
```

- 详见：
  - 【已解决】unicorn代码报错：ERROR Invalid memory write UC_ERR_WRITE_UNMAPPED
- 引申

给UC_ERR_WRITE_UNMAPPED单独加上hook看出错时详情

- 【已解决】unicorn模拟ARM64代码：给UC_ERR_WRITE_UNMAPPED单独加上hook看出错时详情

通过代码：

```
def hook_unmapped(mu, access, address, length, value, context):
    pc = mu.reg_read(UC_ARM64_REG_PC)
    print("! mem unmapped: pc: 0x%X access: %d address: 0x%X length: 0x%x value: 0x%X" %
 (pc, access, address, length, value))
    mu.emu_stop()
    return True

# hook unmamapped memory
mu.hook_add(UC_HOOK_MEM_UNMAPPED, hook_unmapped)
```

实现了一次性hook了，所有类型的unmapped未映射内存的异常

- UC_MEM_READ_UNMAPPED
- UC_MEM_WRITE_UNMAPPED
- UC_MEM_FETCH_UNMAPPED

注：另外想要分别单独去hook，应该也是可以的：

- UC_HOOK_MEM_READ_UNMAPPED
- UC_HOOK_MEM_WRITE_UNMAPPED
- UC_HOOK_MEM_FETCH_UNMAPPED

效果：此处（当出错时）可以输出错误详情：

```
mem unmapped: pc: 0x10000 access: 20 address: 0x7FFF98 length: 0x8 value: 0x0
```

其含义是：

- 当前PC地址：`0x10000`
- 具体操作：`20` == `UC_MEM_WRITE_UNMAPPED`
  - 内存写入时，出现内存未映射的错误
- 具体（此处是写入）操作的地址：`0x7FFF98`
- 具体操作的长度：`8` 个字节
- （此处要写入的）涉及的值：`0`

手动修改指令

如果指令不好模拟，数量不多的话，可以考虑手动修改原始二进制：

举例1：手动替换指令，比如 `braa` 变 `br`

如果此处模拟的代码是 `arm64e` 的，但Unicorn本身暂时又不支持 `arm64e` 的话，对于PAC指令 `braa` 来说，就无法执行

而如果这里指令数量不是很多的话，就可以考虑：手动处理，比如替换成 `br` 。

- 举例

【已解决】Unicorn模拟ARM64代码：手动把braa改为br指令看是否还会报错UC_ERR_EXCEPTION

中的，去把：

- PC=0x0001009C
  - 原先opcode： `31 09 1F D7`
    - `BRAA x9, x17`
  - 改为opcode： `20 01 1F D6`
    - `BR X9`

举例2：其他指令变nop空指令

如果是其他情况，比如某些行的指令，不想要去模拟，则可以直接手动去修改为 `nop` = `空指令` ，而去绕过即可

- 举例：

【已解决】通过修改ARM汇编二进制文件实现Unicorn忽略执行特定指令

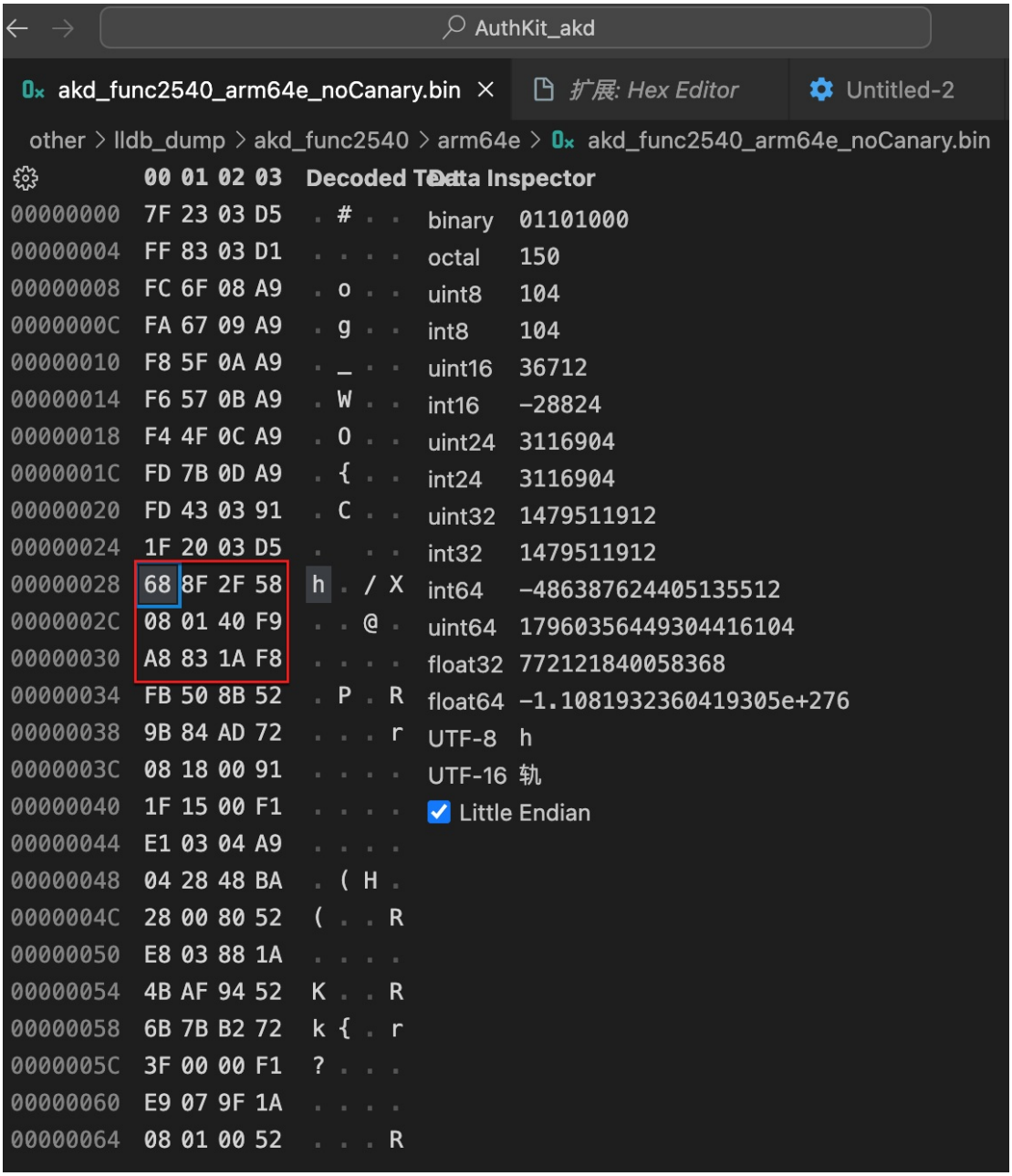
中的：

通过 `VSCode` 的 `Hex Editor` 去修改了，原先的二进制，把

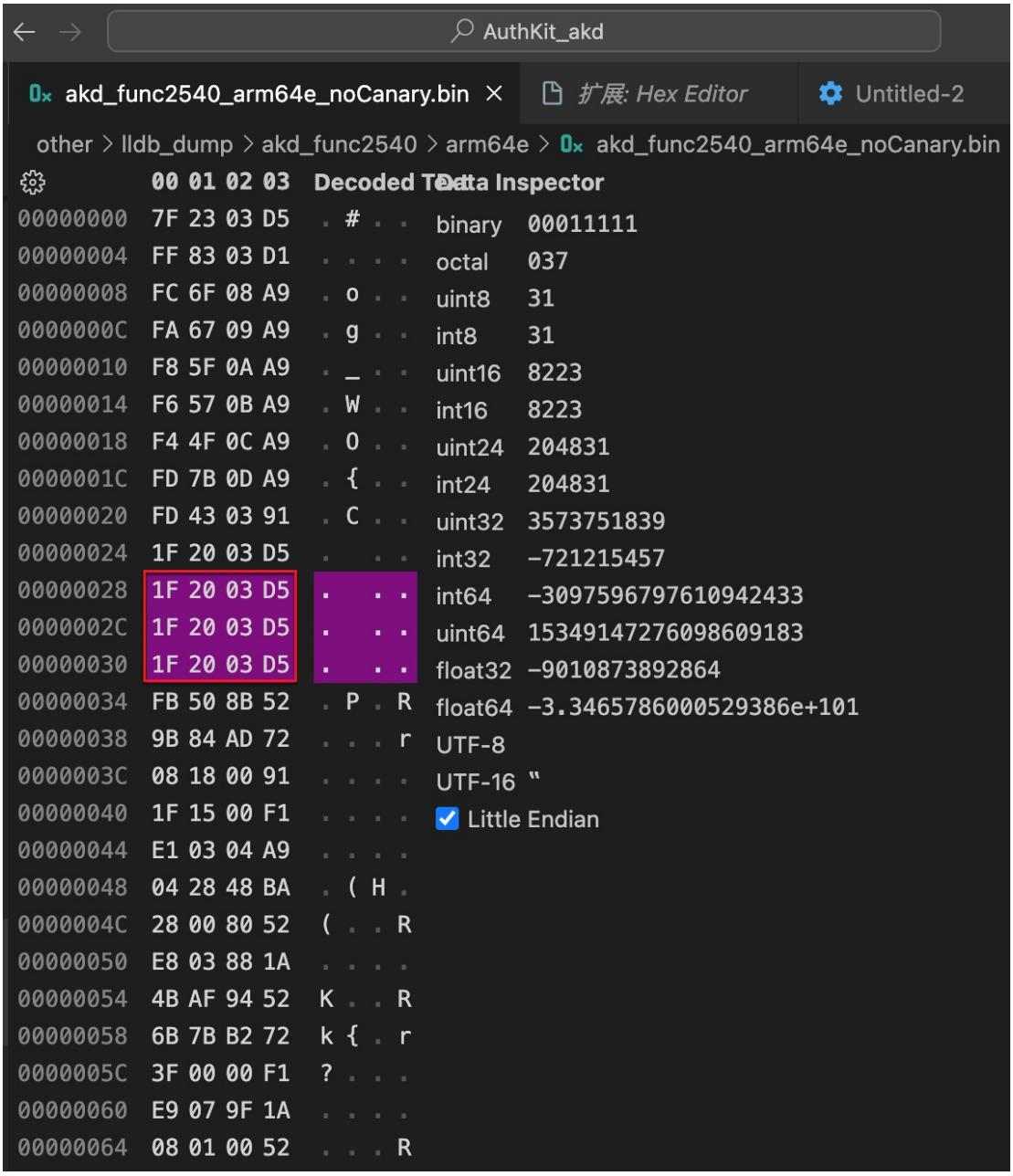
- `__stack_chk_guard`
- `__stack_chk_fail`

相关部分的二进制，直接替换改为了 `NOP` 空指令：

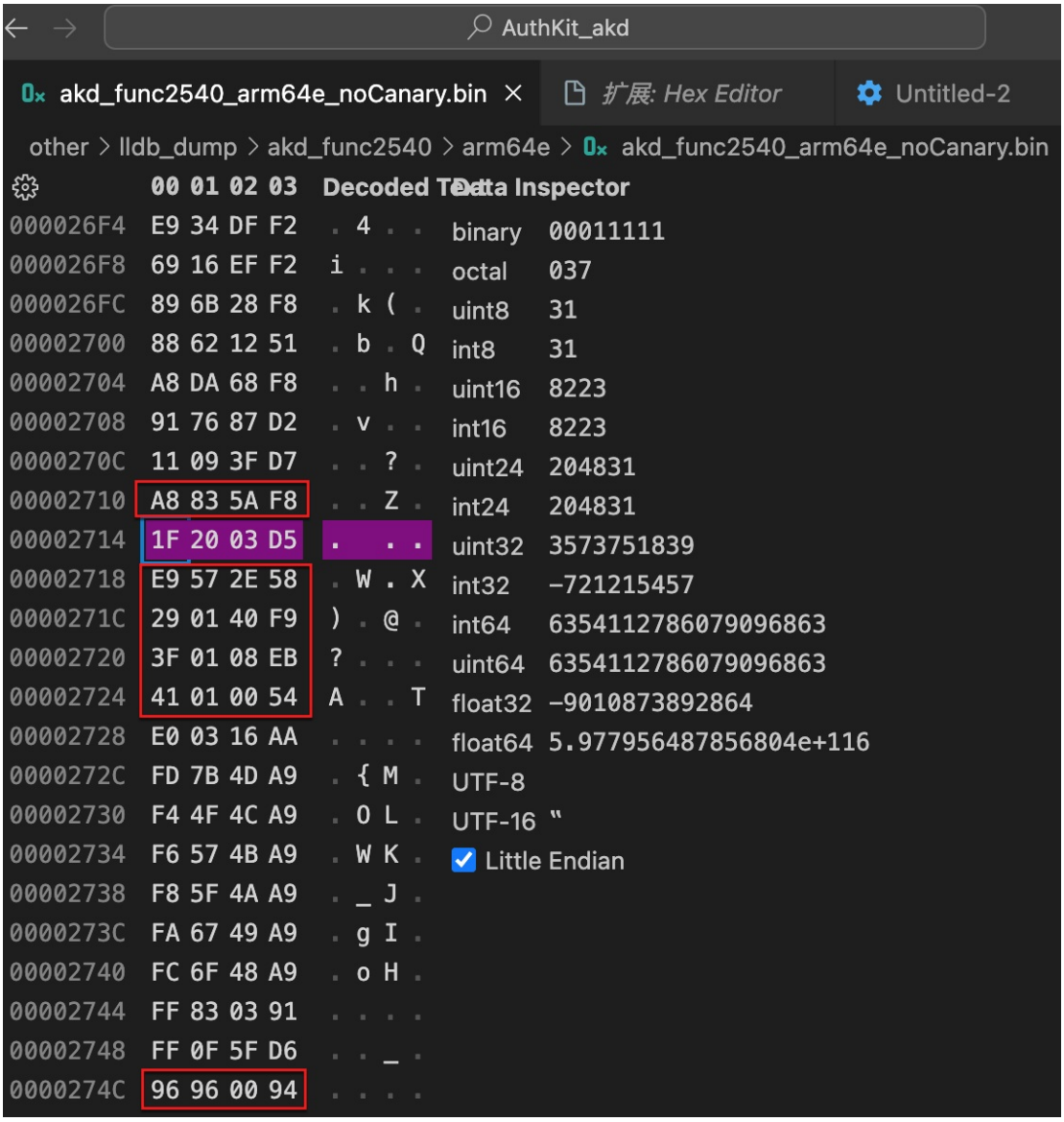
- __stack_chk_guard
  - 改动前


AuthKit_akd
akd_func2540_arm64e_noCanary.bin
扩展: Hex Editor
Untitled-2
other > lldb_dump > akd_func2540 > arm64e > akd_func2540_arm64e_noCanary.bin
00 01 02 03
Decoded Text
Data Inspector
00000000 7F 23 03 D5 . # . .
00000004 FF 83 03 D1 . . . .
00000008 FC 6F 08 A9 . o . .
0000000C FA 67 09 A9 . g . .
00000010 F8 5F 0A A9 . _ . .
00000014 F6 57 0B A9 . W . .
00000018 F4 4F 0C A9 . O . .
0000001C FD 7B 0D A9 . { . .
00000020 FD 43 03 91 . C . .
00000024 1F 20 03 D5 . . . .
00000028 68 8F 2F 58 h . / X
0000002C 08 01 40 F9 . . @ .
00000030 A8 83 1A F8 . . . .
00000034 FB 50 8B 52 . P . R
00000038 9B 84 AD 72 . . . r
0000003C 08 18 00 91 . . . .
00000040 1F 15 00 F1 . . . .
00000044 E1 03 04 A9 . . . .
00000048 04 28 48 BA . (H .
0000004C 28 00 80 52 (. . R
00000050 E8 03 88 1A . . . .
00000054 4B AF 94 52 K . . R
00000058 6B 7B B2 72 k { . r
0000005C 3F 00 00 F1 ? . . .
00000060 E9 07 9F 1A . . . .
00000064 08 01 00 52 . . . R
binary 01101000
octal 150
uint8 104
int8 104
uint16 36712
int16 -28824
uint24 3116904
int24 3116904
uint32 1479511912
int32 1479511912
int64 -486387624405135512
uint64 17960356449304416104
float32 772121840058368
float64 -1.1081932360419305e+276
UTF-8 h
UTF-16 轨
Little Endian

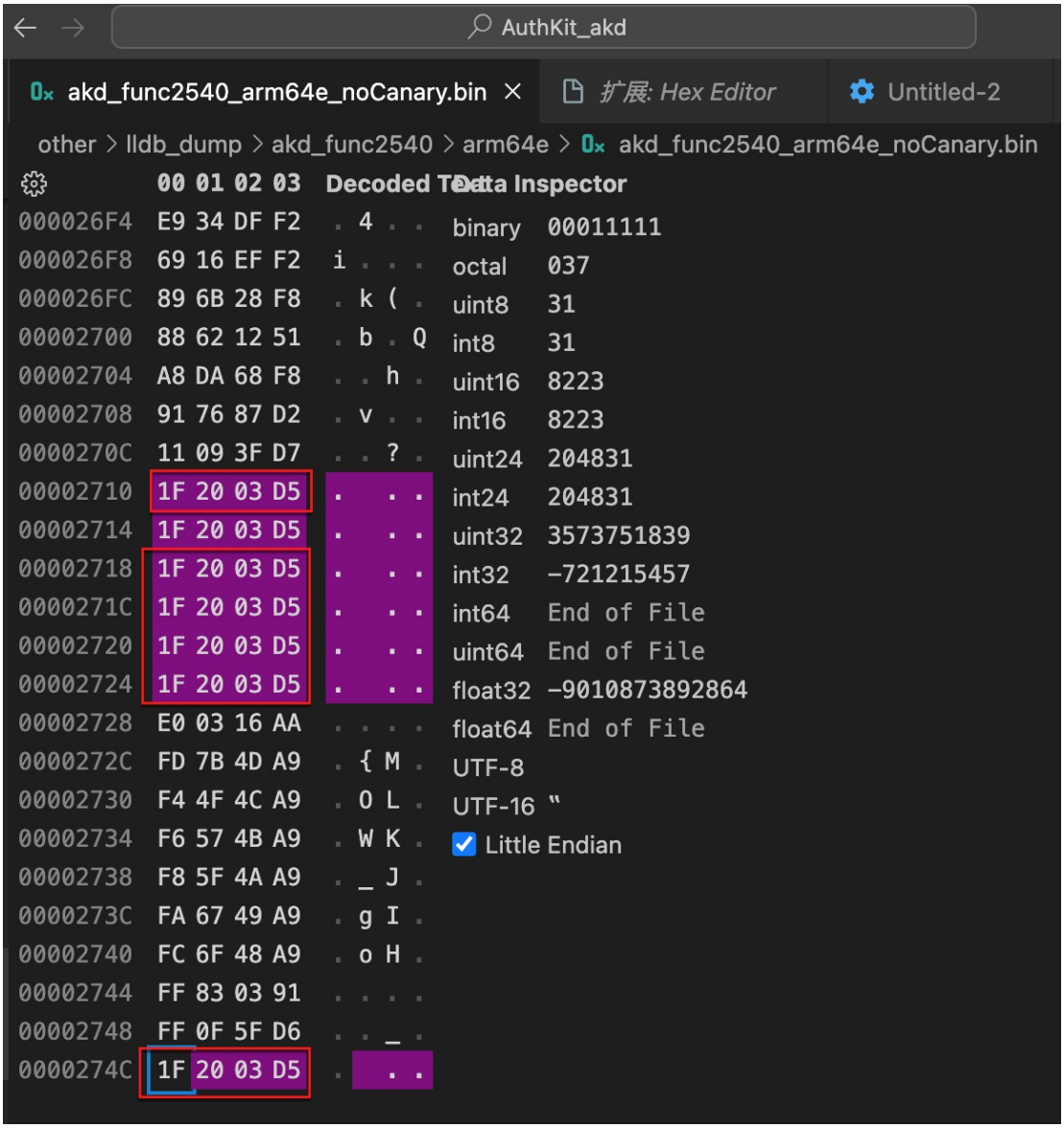
- 改动后



- __stack_chk_fail
  - 改动前


AuthKit_akd
akd_func2540_arm64e_noCanary.bin
扩展: Hex Editor
Untitled-2
other > lldb_dump > akd_func2540 > arm64e > akd_func2540_arm64e_noCanary.bin
00 01 02 03
Decoded Text
Data Inspector
000026F4 E9 34 DF F2 . 4 . .
000026F8 69 16 EF F2 i . . .
000026FC 89 6B 28 F8 . k (.
00002700 88 62 12 51 . b . Q
00002704 A8 DA 68 F8 . . h .
00002708 91 76 87 D2 . v . .
0000270C 11 09 3F D7 . . ? .
00002710 A8 83 5A F8 . . Z .
00002714 1F 20 03 D5 . . .
00002718 E9 57 2E 58 . W . X
0000271C 29 01 40 F9) . @ .
00002720 3F 01 08 EB ? . . .
00002724 41 01 00 54 A . . T
00002728 E0 03 16 AA . . . .
0000272C FD 7B 4D A9 . { M .
00002730 F4 4F 4C A9 . O L .
00002734 F6 57 4B A9 . W K .
00002738 F8 5F 4A A9 . _ J .
0000273C FA 67 49 A9 . g I .
00002740 FC 6F 48 A9 . o H .
00002744 FF 83 03 91 . . . .
00002748 FF 0F 5F D6 . . _ .
0000274C 96 96 00 94 . . . .
binary 00011111
octal 037
uint8 31
int8 31
uint16 8223
int16 8223
uint24 204831
int24 204831
uint32 3573751839
int32 -721215457
int64 6354112786079096863
uint64 6354112786079096863
float32 -9010873892864
float64 5.977956487856804e+116
UTF-8
UTF-16 “
Little Endian

- 改动后



数值转换

在Unicorn模拟期间，往往涉及到，向内存中写入对应的值，以及，从内存中读取出特定的值。

在此期间，往往会涉及到：

- 把对应的数据（int、long等）转换成原始的值，再写入内存
- 把从内存中读取出的raw原始数据，转换成对应的数据类型（int、long等）

此处整理相关的心得：

把数据写入内存

已整理出相关函数，详见：

模拟akd函数symbol2575

中的：`writeMemory`

```
def writeMemory(memAddr, newValue, byteLen):
    """
        for ARM64 little endian, write new value into memory address
        memAddr: memory address to write
        newValue: value to write
        byteLen: 4 / 8
    """
    global uc

    valueFormat = "0x%016X" if byteLen == 8 else "0x%08X"
    if isinstance(newValue, bytes):
        logging.info("writeMemory: memAddr=0x%X, newValue=0x%s, byteLen=%d", memAddr, n
ewValue.hex(), byteLen)
        newValueBytes = newValue
    else:
        valueStr = valueFormat % newValue
        logging.info("writeMemory: memAddr=0x%X, newValue=%s, byteLen=%d", memAddr, val
ueStr, byteLen)
        newValueBytes = newValue.to_bytes(byteLen, "little")
    uc.mem_write(memAddr, newValueBytes)
    logging.info(" >> has write newValueBytes=%s to address=0x%X", newValueBytes, memAd
dr)

    # # for debug: verify write is OK or not
    # readoutValue = uc.mem_read(memAddr, byteLen)
    # logging.info("for address 0x%X, readoutValue hex=0x%s", memAddr, readoutValue.hex
()))
    # # logging.info("readoutValue hexlify=%b", binascii.hexlify(readoutValue))
    # readoutValueLong = int.from_bytes(readoutValue, "little", signed=False)
    # logging.info("readoutValueLong=0x%x", readoutValueLong)
    # # if readoutValue == newValue:
    # if readoutValueLong == newValue:
    #     logging.info("=== Write and read back OK")
    # else:
```

```
    #     logging.info("!!! Write and read back Failed")
```

- 说明

Unicorn模拟期间，常需要，给特定内存地址写入特定的值，用于模拟函数代码的真实的值。

此时，就用调用此函数 `writeMemory` ，给特定内存地址，写入对应的值了。

其中被注释的掉的部分， 恢复后是：

```
    # for debug: verify write is OK or not
    readoutValue = uc.mem_read(memAddr, byteLen)
    logging.info("for address 0x%X, readoutValue hex=0x%s", memAddr, readoutValue.hex())
)
    # logging.info("readoutValue hexlify=%b", binascii.hexlify(readoutValue))
    readoutValueLong = int.from_bytes(readoutValue, "little", signed=False)
    logging.info("readoutValueLong=0x%x", readoutValueLong)
    # if readoutValue == newValue:
    if readoutValueLong == newValue:
        logging.info("=== Write and read back OK")
    else:
        logging.info("!!! Write and read back Failed")
```

可以去：用于写入后，立刻读取出来，验证和写入的值是否一致，验证写入的操作，是否正确。

- 用法举例

```
  writeMemory(0x69C18, 0x0000000000078dfa, 8) # <+4400>: 36 D9 68 F8  -> ldr     x22, [
x9, w8, sxtw #3]
```

就是之前调试了真实的函数后，去给：

- 内存地址： `0x69C18`
  - 注：对应着实际调试期间的 `0x69C18` = `0x59C18` + `0x10000` 中的 `0x59C18` 的相对地址，其中 `0x10000` 是代码的基地址
- 写入对应的值： `0x0000000000078dfa`
- 字节大小=占用地址空间大小（字节数）： `8` 个字节

从内存中读取数据

已整理出相关函数，详见：

模拟akd函数symbol2575

中的： `readMemory`

```
def readMemory(memAddr, byteNum, endian="little", signed=False):
    """read out value from memory"""
    global uc
    readoutRawValue = uc.mem_read(memAddr, byteNum)
    logging.info(" >> readoutRawValue hex=0x%s", readoutRawValue.hex())
```

```
    readoutValue = int.from_bytes(readoutRawValue, endian, signed=signed)
    logging.info(" >> readoutValue=0x%016X", readoutValue)
    return readoutValue
```

- 说明

`Unicorn`中的`mem_read`函数读取出来的，是raw value=原始的值=原始的二进制数据

而往往我们之前保存进去的是，对应的int、long等类型的数据

此时，将raw value转换成int、long等数值时，就可以用此处的`readMemory`

- 用法举例

比如之前写入了对应的值：

```
  writeMemory(0x32850, 0x00000094, 4)             # <+236>: 29 DB A9 B8  -> ldrsw   x9,
[x25, w9, sxtw #2]
```

然后就可以去用readMemory去读取出对应的值：

```
  readMemory(0x32850, 4)
```

用于验证之前写入的值，是否正确。

ARM64和arm64e

Unicorn支持多种架构，其中包括 `ARM`

ARM架构和Unicorn的相关概念有：

- `ARM` ：指的是总体的概念，ARM架构
  - `ARM64` ：指的是，ARM架构下的子架构，64位的 `ARM64`
    - `arm64e` ：指的是，ARM64之后，新增加的， `ARMv8.3` 之后新增了 `PAC` 指令，对应底层ARM汇编成为 `arm64e` ，其支持新的PAC相关指令

由此，要注意的有些细节：

- Unicorn中的ARM
  - `ARM64` 和 `ARM` ，有些寄存器是公共的，所以放到了ARM中，而ARM64没有
    - 比如
      - 有： `UC_ARM_REG_CPSR`
      - 没有： `UC_ARM64_REG_CPSR`
  - 暂时不支持 `arm64e` （的PAC指令）
    - 举例
      - pacibsp

        ```
        akd`___lldb_unnamed_symbol2540$$akd:
        ->  0x1045f598c <+0>:  pacibsp
        ```

    - 所以无法彻底解决
      - 会报错：UC_ERR_EXCEPTION
        - 举例
          - 不支持arm64e的pac指令BRAA而导致CPU异常： `ERROR Unhandled CPU exception UC_ERR_EXCEPTION`
      - 只能用其他办法规避
        - 比如
          - 把PAC相关指令，换成NOP空指令或对应的去掉PAC部分的指令
            - 详见：手动修改指令

调用其他子函数

Unicorn模拟某个函数运行期间，被模拟的函数A，往往会调用到其他函数B，函数C等等。此时，就涉及到：

Unicorn中，模拟调用 `子函数` = `其他函数` 。

模拟调用子函数的思路和框架

Unicorn中模拟ARM64代码，去模拟A函数：

遇到 `blr x8` 跳转到B函数，去模拟B函数，搭建一个空的框架，供B函数使用。

此处总体思路是：

- 弄出一个框架函数
  - 暂时只有一个ARM64的 `little endian` 的 `ret` 指令
    - 对应值，可以自己手动推算或借助在线网站
      - Online ARM to HEX Converter (armconverter.com)
    - 帮忙算出来是： `0x C0 03 5F D6`
      - 对应写成Python二进制就是： `b"\xC0\x03\x5F\xD6"`
    - 注：当然，你可以根据自己需要，去加上更多行的代码
      - ARM汇编转opcode二进制，可以参考上述在线网站去生成
  - 而后续会去给该函数的代码加上 `hook_code`
    - 加上相关的逻辑：
      - 获取传入的参数：比如ARM中的**第一个参数** `x0` 的值
      - 加上对应处理逻辑：比如此处只是用于演示demo用：给 `x0` 加上 `100`
        - 后续可以根据需要，变成自己的处理逻辑
          - 比如想办法用Python代码实现 `malloc` 的效果，返回真正的新申请的内存的地址
      - 用 `x0` 返回值：把新的值写入 `x0` 寄存器
- 再去把上述的框架函数的opcode，写入对应的内存地址
  - 作为ARM64的代码，用于后续跳转后执行
- 然后去给对应 `blr x8` 对应的地址，去写入对应的上述新的框架函数的地址，即可

举例：模拟调用malloc

此处，用于框架代码，后续用于模拟malloc的函数，暂且叫做 emulateMalloc 相关的实际代码是：

```
uc = None

#------------------- emulate malloc --------------------
emulateMallocOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateMallocCodeSize = len(emulateMallocOpcode)
EMULATE_MALLOC_CODE_START = 2 * 1024 * 1024
EMULATE_MALLOC_CODE_END = EMULATE_MALLOC_CODE_START + gEmulateMallocCodeSize
```

```
def writeMemory(memAddr, newValue, byteLen):
    """
        for ARM64 little endian, write new value into memory address
        memAddr: memory address to write
        newValue: value to write
        byteLen: 4 / 8
    """
    global uc

    valueFormat = "0x%016X" if byteLen == 8 else "0x%08X"
    if isinstance(newValue, bytes):
        print("writeMemory: memAddr=0x%X, newValue=0x%s, byteLen=%d" % (memAddr, newVal
ue.hex(), byteLen))
        newValueBytes = newValue
    else:
        valueStr = valueFormat % newValue
        print("writeMemory: memAddr=0x%X, newValue=%s, byteLen=%d" % (memAddr, valueStr,
 byteLen))
        newValueBytes = newValue.to_bytes(byteLen, "little")
    uc.mem_write(memAddr, newValueBytes)
    print(" >> has write newValueBytes=%s to address=0x%X" % (newValueBytes, memAddr))


# callback for tracing instructions
def hook_code(mu, address, size, user_data):
    pc = mu.reg_read(UC_ARM64_REG_PC)
...
    # common debug
...
    x0 = mu.reg_read(UC_ARM64_REG_X0)
    x1 = mu.reg_read(UC_ARM64_REG_X1)
...

    # for emulateMalloc
    if pc == 0x00200000:
        # emulate pass in parameter(s)/argument(s)
        curX0 = mu.reg_read(UC_ARM64_REG_X0)
        # emulate do something: here is add 100
        retValue = curX0 + 100
        # emulate return value
        mu.reg_write(UC_ARM64_REG_X0, retValue)
        print("input x0=0x%x, output ret: 0x%x" % (curX0, retValue))

# Emulate arm function running
def emulate_akd_arm64e_symbol2540():
    global uc
    print("Emulate arm64 sub_1000A0460 == ___lldb_unnamed_symbol2575$$akd function runn
ing")
    try:
        mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM + UC_MODE_LITTLE_ENDIAN)
...
        # for emuleateMalloc
        writeMemory(EMULATE_MALLOC_CODE_START, emulateMallocOpcode, gEmulateMallocCodeS
ize)
        writeMemory(0x69BD8, EMULATE_MALLOC_CODE_START + 2, 8)
```

即可输出对应期望的内容：

```
=== 0x000100C8   +200 : E8 33 00 F9  -> str     x8, [sp, #0x60]
 >> Memory WRITE at 0x77FF70, size 8, value 0x200000, PC 0x100C8
=== 0x000100CC   +204 : 00 01 3F D6  -> blr     x8
>>> Tracing basic block at 0x200000, block size = 0x4
=== 0x00200000  +2031616 : C0 03 5F D6  -> ret
    debug: PC 0x200000: cpsr 0x20000000, x0 0x0000000000000018, x1 0x0000000000410000
input x0 0x18, output ret: 0x7c
>>> Tracing basic block at 0x100d0, block size = 0x50
=== 0x000100D0   +208 : 08 00 80 52  -> movz    w8, #0
    debug: PC 0x100D0: x0 0x000000000000007C, x1 0x0000000000410000
=== 0x000100D4   +212 : 1F 00 00 F1  -> cmp     x0, #0
```

输出的log对应的逻辑解释：

即从原先的代码：

```
<+204> : 00 01 3F D6   -> blr     x8
```

跳转到了，我此处的 `emulateMalloc` 的函数的地址 `0x00200000` ，去运行了

此函数中，暂时只有一行的ARM64代码：

```
0x00200000  +2031616 : C0 03 5F D6  -> ret
```

其中传入的参数：

- `x0 = 0x0000000000000018`

经过自己hook_code中的处理后：

- 返回值= `x0 = 0x7c`

然后代码返回原先代码的下一行：

```
0x000100D0  <+208 : 08 00 80 52   -> movz    w8, #0
```

继续去运行，且对应的返回值：

- `x0 = 0x000000000000007C`

是符合预期的，是我们故意返回的值。

如此，模拟一个B（的框架）函数，供A函数去调用和跳转后再返回，就完成了。

- 附录

完整代码详见：

模拟akd函数symbol2575

其中就有真正的完整的模拟malloc的代码，供参考。

模拟函数实现

此处接着说，模拟调用其他子函数期间，常常会涉及到的一些，相对通用的函数，如何去模拟。

- 注：完整代码 模拟akd函数symbol2575 ，包括下面几个模拟函数，需要的可以去参考。

模拟malloc申请内存

此处最后是：

- 参考网上源码unicorn_loader
- 加上自己后续的优化

目前最新代码是：

- `UnicornSimpleHeap.py`

```
# Function: Emulate memory management (malloc/free/...)
# Author: Crifan Li
# Update: 20230529

from unicorn import *
import logging

# Page size required by Unicorn
UNICORN_PAGE_SIZE = 0x1000

# Max allowable segment size (1G)
MAX_ALLOWABLE_SEG_SIZE = 1024 * 1024 * 1024

# Alignment functions to align all memory segments to Unicorn page boundaries (4KB page
s only)
ALIGN_PAGE_DOWN = lambda x: x & ~(UNICORN_PAGE_SIZE - 1)
ALIGN_PAGE_UP   = lambda x: (x + UNICORN_PAGE_SIZE - 1) & ~(UNICORN_PAGE_SIZE 1)

# refer: https://github.com/Battelle/afl-unicorn/blob/master/unicorn_mode/helper_script
s/unicorn_loader.py
class UnicornSimpleHeap(object):
    """ Use this class to provide a simple heap implementation. This should
        be used if malloc/free calls break things during emulation. This heap also
        implements basic guard-page capabilities which enable immediate notice of
        heap overflow and underflows.
    """

    # Helper data-container used to track chunks
    class HeapChunk(object):
        def __init__(self, actual_addr, total_size, data_size):
            self.total_size = total_size                        # Total size of the chu
nk (including padding and guard page)
            self.actual_addr = actual_addr                      # Actual start address
of the chunk
            self.data_size = data_size                          # Size requested by the
 caller of actual malloc call
```

```
            self.data_addr = actual_addr + UNICORN_PAGE_SIZE    # Address where data ac
tually starts

        # Returns true if the specified buffer is completely within the chunk, else fal
se
        def is_buffer_in_chunk(self, addr, size):
            if addr >= self.data_addr and ((addr + size) <= (self.data_addr + self.data
_size)):
                return True
            else:
                return False

        def isSameChunk(self, anotherChunk):
            isSame = (self.actual_addr == anotherChunk.actual_addr) and (self.total_siz
e == anotherChunk.total_size)
            return isSame

        def debug(self):
            chunkEndAddr = self.actual_addr + self.total_size
            chunkStr = "chunk: [0x%X-0x%X] ptr=0x%X, size=%d=0x%X"% (self.actual_addr,
chunkEndAddr, self.data_addr, self.data_size, self.data_size)
            return chunkStr

        def isOverlapped(self, newChunk):
            # logging.info("debug: self=%s, newChunk=%s", self.debug(), newChunk.debug(
))
            selfStartAddr = self.actual_addr
            selfLastAddr = selfStartAddr + self.total_size - 1
            newChunkStartAddr = newChunk.actual_addr
            newChunkLastAddr = newChunkStartAddr + newChunk.total_size - 1
            isOverlapStart = (newChunkStartAddr >= selfStartAddr) and (newChunkStartAdd
r <= selfLastAddr)
            isOverlapEnd = (newChunkLastAddr >= selfStartAddr) and (newChunkLastAddr <=
 selfLastAddr)
            isOverlapped = isOverlapStart or isOverlapEnd
            return isOverlapped

    # # Skip the zero-page to avoid weird potential issues with segment registers
    # HEAP_MIN_ADDR = 0x00002000 # 8KB
    # HEAP_MAX_ADDR = 0xFFFFFFFF # 4GB-1
    _headMinAddr = None
    _heapMaxAddr = None

    _uc = None              # Unicorn engine instance to interact with
    _chunks = []            # List of all known chunks
    _debug_print = False    # True to print debug information

    # def __init__(self, uc, debug_print=False):
    def __init__(self, uc, headMinAddr, heapMaxAddr, debug_print=False):
        self._uc = uc
        self._headMinAddr = headMinAddr
        self._heapMaxAddr = heapMaxAddr
        self._debug_print = debug_print

        # Add the watchpoint hook that will be used to implement psuedo-guard page supp
ort
```

```
        self._uc.hook_add(UC_HOOK_MEM_WRITE | UC_HOOK_MEM_READ, self.__check_mem_access)


    def isChunkAllocated(self, newChunk):
        isAllocated = False
        for eachChunk in self._chunks:
            if eachChunk.isSameChunk(newChunk):
                isAllocated = True
                break
        return isAllocated

    def isChunkOverlapped(self, newChunk):
        isOverlapped = False
        for eachChunk in self._chunks:
            if eachChunk.isOverlapped(newChunk):
                isOverlapped = True
                break
        return isOverlapped

    def malloc(self, size):
        # Figure out the overall size to be allocated/mapped
        #    - Allocate at least 1 4k page of memory to make Unicorn happy
        #    - Add guard pages at the start and end of the region
        total_chunk_size = UNICORN_PAGE_SIZE + ALIGN_PAGE_UP(size) + UNICORN_PAGE_SIZE
        # Gross but efficient way to find space for the chunk:
        chunk = None
        # for addr in range(self.HEAP_MIN_ADDR, self.HEAP_MAX_ADDR, UNICORN_PAGE_SIZE):
        for addr in range(self._headMinAddr, self._heapMaxAddr, UNICORN_PAGE_SIZE):
            try:
                # self._uc.mem_map(addr, total_chunk_size, UC_PROT_READ | UC_PROT_WRITE)

                chunk = self.HeapChunk(addr, total_chunk_size, size)
                # chunkStr = "[0x{0:X}-0x{1:X}]".format(chunk.actual_addr, chunk.actual
_addr + chunk.total_size)
                chunkStr = chunk.debug()
                # if chunk in self._chunks:
                # if self.isChunkAllocated(chunk):
                if self.isChunkOverlapped(chunk):
                    # if self._debug_print:
                    #     logging.info(" ~~ Omit overlapped chunk: %s", chunkStr)
                    continue
                else:
                    if self._debug_print:
                        # logging.info("Heap: allocating 0x{0:X} byte addr=0x{1:X} of c
hunk {2:s}".format(chunk.data_size, chunk.data_addr, chunkStr))
                        logging.info(" ++ Allocated heap chunk: %s", chunkStr)
                    break
            except UcError as err:
                logging.error("!!! Heap malloc failed: error=%s", err)
                continue
        # Something went very wrong
        if chunk == None:
            return 0
        self._chunks.append(chunk)
        return chunk.data_addr
```

```
    def calloc(self, size, count):
        # Simple wrapper around malloc with calloc() args
        return self.malloc(size count)

    def realloc(self, ptr, new_size):
        # Wrapper around malloc(new_size) / memcpy(new, old, old_size) / free(old)
        if self._debug_print:
            logging.info("Reallocating chunk @ 0x{0:016x} to be 0x{1:x} bytes".format(p
tr, new_size))
        old_chunk = None
        for chunk in self._chunks:
            if chunk.data_addr == ptr:
                old_chunk = chunk
        new_chunk_addr = self.malloc(new_size)
        if old_chunk != None:
            self._uc.mem_write(new_chunk_addr, str(self._uc.mem_read(old_chunk.data_addr
, old_chunk.data_size)))
            self.free(old_chunk.data_addr)
        return new_chunk_addr

    def free(self, addr):
        for chunk in self._chunks:
            if chunk.is_buffer_in_chunk(addr, 1):
                if self._debug_print:
                    logging.info("Freeing 0x{0:x}-byte chunk @ 0x{0:016x}".format(chunk.
req_size, chunk.data_addr))
                self._uc.mem_unmap(chunk.actual_addr, chunk.total_size)
                self._chunks.remove(chunk)
                return True
        return False

    # Implements basic guard-page functionality
    def __check_mem_access(self, uc, access, address, size, value, user_data):
        for chunk in self._chunks:
            if address >= chunk.actual_addr and ((address + size) <= (chunk.actual_addr
+ chunk.total_size)):
                if chunk.is_buffer_in_chunk(address, size) == False:
                    if self._debug_print:
                        logging.info("Heap over/underflow attempting to {0} 0x{1:x} byt
es @ {2:016x}".format( \
                            "write" if access == UC_MEM_WRITE else "read", size, address
))
                    # Force a memory-based crash
                    uc.force_crash(UcError(UC_ERR_READ_PROT))
```

调用代码：`emulate_akd_getIDMSRoutingInfo.py`

```
from libs.UnicornSimpleHeap import UnicornSimpleHeap

ucHeap = None

#------------------- Stack -------------------
HEAP_ADDRESS = 6 * 1024 * 1024
HEAP_SIZE = 1 * 1024 * 1024
```

```
HEAP_ADDRESS_END = HEAP_ADDRESS + HEAP_SIZE
HEAP_ADDRESS_LAST_BYTE = HEAP_ADDRESS_END - 1

...
# callback for tracing instructions
def hook_code(mu, address, size, user_data):
    global ucHeap

    # for emulateMalloc
    if pc == 0x00200000:
        mallocSize = mu.reg_read(UC_ARM64_REG_X0)
        newAddrPtr = ucHeap.malloc(mallocSize)
        mu.reg_write(UC_ARM64_REG_X0, newAddrPtr)
        print("input x0=0x%x, output ret: 0x%x" % (mallocSize, newAddrPtr))

...
def emulate_arm64():
    global uc, ucHeap
    try:
        # Initialize emulator in ARM mode
        # mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)
        mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM + UC_MODE_LITTLE_ENDIAN)
...
        # map heap
        mu.mem_map(HEAP_ADDRESS, HEAP_SIZE)
        print("Mapped memory: Heap\t[0x%016X-0x%016X]" % (HEAP_ADDRESS, HEAP_ADDRESS +
HEAP_SIZE))
```

输出：

```
Heap: allocating 0x18-byte chunk @ 0x0000000000601000
input x0 0x18, output ret: 0x601000
```

模拟malloc分配出内存，供后续使用。

模拟free释放内存

此处模拟free去释放内存，其实就是参考自己的模拟调用子函数的框架，然后加上，其实就一行代码 `ret` 而已。

相关部分代码是：

```
#------------------- emulate free --------------------
emulateFreeOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateFreeCodeSize = len(emulateFreeOpcode)


EMULATE_FREE_CODE_START = (2 * 1024 * 1024) + (128 * 1024)
EMULATE_FREE_CODE_END = EMULATE_FREE_CODE_START + gEmulateFreeCodeSize


FREE_JUMP_ADDR = 0x69B88
FREE_JUMP_VALUE = EMULATE_FREE_CODE_START + 2
```

```
FREE_JUMP_SIZE = 8

...

def hook_code(mu, address, size, user_data):
...
    if pc == EMULATE_FREE_CODE_START:
        address = mu.reg_read(UC_ARM64_REG_X0)
        print("emulateFree: input address=0x%x" % (address))
...


print("\t\t\t  [0x%08X-0x%08X]   emulateFree jump" % (FREE_JUMP_ADDR, FREE_JUMP_ADDR +
FREE_JUMP_SIZE))

print("\t\t\t  [0x%08X-0x%08X] func: emulateFree" % (EMULATE_FREE_CODE_START, EMULATE_F
REE_CODE_END))

        # for emuleateFree
        writeMemory(EMULATE_FREE_CODE_START, emulateFreeOpcode, gEmulateFreeCodeSize)
        writeMemory(FREE_JUMP_ADDR, FREE_JUMP_VALUE, FREE_JUMP_SIZE) # <+256>: 0A DB 6A
 F8  -> ldr     x10, [x24, w10, sxtw #3]
```

log输出是：

```
Mapped memory: Code     [0x00010000-0x00410000]
                         [0x00010000-0x000124C8] func: ___lldb_unnamed_symbol2575$$akd
                         [0x00031220-0x00033450]   fix br err: x9SmallOffset
                         [0x00068020-0x00069B80]   fix br err: x10AbsFuncAddrWithOffse
t
                         [0x00069B88-0x00069B90]   emulateFree jump
                         [0x00069BC0-0x00069BC8]   emulateAkdFunc2567 jump
                         [0x00069BD8-0x00069BE0]   emulateMalloc jump
                         [0x00069BE8-0x00069BF0]   line 7392 jump
                         [0x00069C08-0x00069C10]   emulateDemalloc jump
                         [0x00200000-0x00200004] func: emulateMalloc
                         [0x00220000-0x00220004] func: emulateFree
                         [0x00280000-0x00280004] func: emulateAkdFunc2567
Mapped memory: Libc     [0x00500000-0x00580000]
Mapped memory: Heap     [0x00600000-0x00700000]
Mapped memory: Stack    [0x00700000-0x00800000]
Mapped memory: Args     [0x00800000-0x00810000]
...
writeMemory: memAddr=0x220000, newValue=0xc0035fd6, byteLen=4
 >> has write newValueBytes=b'\xc0\x03_\xd6' to address=0x220000
writeMemory: memAddr=0x69B88, newValue=0x0000000000220002, byteLen=8
 >> has write newValueBytes=b'\x02\x00"\x00\x00\x00\x00\x00' to address=0x69B88
...
===  0x00010100   +256 : 0A DB 6A F8  -> ldr     x10, [x24, w10, sxtw #3]
     debug: PC=0x10100: x24=0x0000000000069B80, w10=0x00000001
 <<  Memory READ at 0x69B88, size=8, rawValueLittleEndian=0x0200220000000000, pc=0x10100
===  0x00010104   +260 : 5B 09 00 D1  -> sub     x27, x10, #2
     debug: PC=0x10104: x10=0x0000000000220002
===  0x00010108   +264 : 56 F9 95 12  -> movn    w22, #0xafca
     debug: PC=0x10108: x27=0x0000000000220000
```

之后输出：

```
=== 0x0001244C +9292 : F3 03 10 AA  -> mov     x19, x16
    debug: PC 0x1244C: x16 0xD5709BDDEAB9B930
=== 0x00012450 +9296 : 60 03 3F D6  -> blr     x27
    debug: PC 0x12450: x27 0x0000000000220000
>>> Tracing basic block at 0x220000, block size = 0x4
=== 0x00220000 +2162688 : C0 03 5F D6  -> ret
emulateFree: input address=0x604000
```

跳转到了此处的 `free` ，且传入的地址，是之前 `malloc` 出来的地址： `0x604000` ，即可。

模拟vm_deallocate释放内存

最后去用代码模拟demalloc释放内存：

```
def readMemory(memAddr, byteNum, endian="little", signed=False):
    """read out value from memory"""
    global uc
    readoutRawValue = uc.mem_read(memAddr, byteNum)
    print(" >> readoutRawValue hex=0x%s" % readoutRawValue.hex())
    readoutValueLong = int.from_bytes(readoutRawValue, endian, signed signed)
    print(" >> readoutValueLong=0x%016X" % readoutValueLong)
    return readoutValueLong

def writeMemory(memAddr, newValue, byteLen):
    """
        for ARM64 little endian, write new value into memory address
        memAddr: memory address to write
        newValue: value to write
        byteLen: 4 / 8
    """
    global uc

    valueFormat = "0x%016X" if byteLen == 8 else "0x%08X"
    if isinstance(newValue, bytes):
        print("writeMemory: memAddr=0x%X, newValue=0x%s, byteLen=%d" % (memAddr, newVal
ue.hex(), byteLen))
        newValueBytes = newValue
    else:
        valueStr = valueFormat % newValue
        print("writeMemory: memAddr=0x%X, newValue=%s, byteLen=%d" % (memAddr, valueStr,
 byteLen))
        newValueBytes = newValue.to_bytes(byteLen, "little")
    uc.mem_write(memAddr, newValueBytes)
    print(" >> has write newValueBytes=%s to address=0x%X" % (newValueBytes, memAddr))

#------------------- emulate demalloc -------------------
emulateDemallocOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateDemallocCodeSize = len(emulateDemallocOpcode)

EMULATE_DEMALLOC_CODE_START = (2 * 1024 * 1024) + (256 * 1024)
```

```
EMULATE_DEMALLOC_CODE_END = EMULATE_DEMALLOC_CODE_START + gEmulateDemallocCodeSize

DEMALLOC_JUMP_ADDR = 0x69C08
DEMALLOC_JUMP_VALUE = EMULATE_DEMALLOC_CODE_START + 2
DEMALLOC_JUMP_SIZE = 8

...

def hook_code(mu, address, size, user_data):
...
    if pc == EMULATE_DEMALLOC_CODE_START:
        targetTask = mu.reg_read(UC_ARM64_REG_X0)
        address = mu.reg_read(UC_ARM64_REG_X1)
        size = mu.reg_read(UC_ARM64_REG_X2)
        # zeroValue = 0
        # zeroValueBytes = zeroValue.to_bytes(size, "little")
        if (address > 0) and (size > 0):
            writeMemory(address, 0, size)
        print("emulateDemalloc: input targetTask=0x%X,address=0x%X,size=%d=0x%X" % (tar
getTask, address, size, size))
        gNoUse = 1
...
        print("\t\t\t  [0x%08X-0x%08X]   emulateDemalloc jump" % (DEMALLOC_JUMP_ADDR, D
EMALLOC_JUMP_ADDR + DEMALLOC_JUMP_SIZE))
...
        # for emuleateDemalloc
        writeMemory(EMULATE_DEMALLOC_CODE_START, emulateDemallocOpcode, gEmulateDemallo
cCodeSize)
        writeMemory(DEMALLOC_JUMP_ADDR, DEMALLOC_JUMP_VALUE, DEMALLOC_JUMP_SIZE) # <+74
20>: 28 D9 68 F8  -> ldr     x8, [x9, w8, sxtw #3]
```

即可去模拟demalloc去释放内存：

此处只是设置对应内存地址范围内的值都是0

此处输出log：

```
===  0x00011CE0  +7392 : 28 D9 68 F8  ->  ldr     x8, [x9, w8, sxtw #3]
    debug: PC 0x11CE0: w8 0x0000000D, x9 0x0000000000069B80
 <<  Memory READ at 0x69BE8, size 8, rawValueLittleEndian 0x0200080000000000, pc 0x11CE0
===  0x00011CE4  +7396 : 00 E1 5F B8  ->  ldur    w0, [x8, #-2]
    debug: PC 0x11CE4: x8 0x0000000000080002
 <<  Memory READ at 0x80000, size 4, rawValueLittleEndian 0x03020000, pc 0x11CE4
===  0x00011CE8  +7400 : E1 3B 40 F9  ->  ldr     x1, [sp, #0x70]
 <<  Memory READ at 0x77FF80, size 8, rawValueLittleEndian 0x0000000000000000, pc 0x11CE
8
===  0x00011CEC  +7404 : E2 6F 40 B9  ->  ldr     w2, [sp, #0x6c]
 <<  Memory READ at 0x77FF7C, size 4, rawValueLittleEndian 0x00000000, pc 0x11CEC
===  0x00011CF0  +7408 : 48 3F 00 51  ->  sub     w8, w26, #0xf
===  0x00011CF4  +7412 : FA 50 8B 52  ->  movz    w26, #0x5a87
===  0x00011CF8  +7416 : 9A 84 AD 72  ->  movk    w26, #0x6c24, lsl #16
===  0x00011CFC  +7420 : 28 D9 68 F8  ->  ldr     x8, [x9, w8, sxtw #3]
    debug: PC 0x11CFC: w8 0x00000011, x9 0x0000000000069B80
 <<  Memory READ at 0x69C08, size 8, rawValueLittleEndian 0x0200240000000000, pc 0x11CFC
```

```
=== 0x00011D00  +7424 : 08 09 00 D1  -> sub     x8, x8, #2
    debug: PC 0x11D00: x8 0x0000000000240002
=== 0x00011D04  +7428 : 00 01 3F D6  -> blr     x8
>>> Tracing basic block at 0x240000, block size = 0x4
=== 0x00240000  +2293760 : C0 03 5F D6  -> ret
emulateDemalloc: input targetTask 0x203,address 0x0,size 0 0x0
>>> Tracing basic block at 0x11d08, block size = 0x50
=== 0x00011D08  +7432 : F1 43 45 A9  -> ldp     x17, x16, [sp, #0x50]
  << Memory READ at 0x77FF60, size 8, rawValueLittleEndian 0x0010600000000000, pc 0x11D0
8
  << Memory READ at 0x77FF68, size 8, rawValueLittleEndian 0x30b9b9eadd9b70d5, pc 0x11D0
8
...
```

即可顺利继续运行。

其中此处：

- `address = 0x0`
- `size = 0 = 0x0`

-》导致效果是：实际上没有去清空内存值为 `0`

-》原因是：

- `address = 0x0 = x1 = [sp + 0x70]`
- `size = 0 = 0x0 = w2 = [sp + -0x6C]`

对应的sp堆栈的位置中，没有去设置对应的值，所以都是0

用到Unicorn的

- 用到Unicorn的第三方工具/库、别的用途等
  - 据说：可以用Unicorn辅助去混淆
  - Android中利用Unicorn
    - frida_dump
      - https://github.com/lasting-yang/frida_dump
    - 安卓逆向：AndroidNativeEmu
      - https://github.com/AeonLucid/AndroidNativeEmu
        - https://github.com/P4nda0s/AndroidNativeEmu
  - unidbg
    - Github
      - https://github.com/zhkl0228/unidbg
  - Qiling Framework
    - https://qiling.io/
      - Cross platform and multi arch ultra lightweight emulator
  - 其他的，详见官网的showcase
    - https://www.unicorn-engine.org/showcase/

实例

下面整理出一些的Unicorn的实例，供参考。

模拟akd函数symbol2575

此处整个项目的文件架构

```
➜  unicorn_akd git:(master) ✗ pwd
/Users/crifan/xxx/dynamicDebug/emulate_arm/unicorn_akd

➜  unicorn_akd git:(master) ✗ tree .
.
├── debug
│   └── log
├── emulate_akd_getIDMSRoutingInfo.py
├── input
│   └── akd_getIDMSRoutingInfo
│       ├── arm64
│       │   ├── akd_arm64_symbol2575.bin
├── libs
│   ├── UnicornSimpleHeap.py
│   └── crifan
│       └── crifanLogging.py
```

输入文件：`akd_arm64_symbol2575.bin`

下载地址：akd_arm64_symbol2575.bin

核心代码：`emulate_akd_getIDMSRoutingInfo.py`

代码解释

在贴出代码 `emulate_akd_getIDMSRoutingInfo` 之前，先给出代码的解释，详见之前的各个章节：

- 运行前
  - 内存布局
  - 设置代码
  - 函数参数
  - 相关数据
  - Stack栈
  - Heap堆
- 运行中
  - 开始运行
  - 调试逻辑
    - hook
      - hook代码
      - hook内存
      - hook异常
    - 日志

代码

```
# Function: Use Unicorn to emulate akd +[AKADIProxy getIDMSRoutingInfo:forDSID:] intern
al implementation function code to running
#   arm64e: ___lldb_unnamed_symbol2540$$akd
#   arm64: ___lldb_unnamed_symbol2575$$akd
# Author: Crifan Li
# Update: 20220608

from __future__ import print_function
import re
from unicorn import *
from unicorn.arm64_const import *
from unicorn.arm_const import *
# import binascii
from capstone import *
from capstone.arm64 import *

from libs.UnicornSimpleHeap import UnicornSimpleHeap

import os
from datetime import datetime,timedelta
import logging
from libs.crifan import crifanLogging

def getCurDatetimeStr(outputFormat="%Y%m%d_%H%M%S"):
    """
    get current datetime then format to string


    eg:
        20171111_220722


    :param outputFormat: datetime output format
    :return: current datetime formatted string
    """
    curDatetime = datetime.now() # 2017-11-11 22:07:22.705101
    curDatetimeStr = curDatetime.strftime(format=outputFormat) #'20171111_220722'
    return curDatetimeStr


def getFilenameNoPointSuffix(curFilePath):
    """Get current filename without point and suffix


    Args:
        curFilePath (str): current file path. Normally can use __file__
```

```
    Returns:
        str, file name without .xxx
    Raises:
    Examples:
        input: /Users/xxx/pymitmdump/mitmdumpOtherApi.py
        output: mitmdumpOtherApi
    """
    root, pointSuffix = os.path.splitext(curFilePath)
    curFilenameNoSuffix = root.split(os.path.sep)[-1]
    return curFilenameNoSuffix

################################################################################
# Global Variable
################################################################################

# current all code is 4 byte -> single line arm code
# gSingleLineCode = True

# only for debug
gNoUse = 0

BYTES_PER_LINE = 4

uc = None
ucHeap = None

################################################################################
# Util Function
################################################################################

def readBinFileBytes(inputFilePath):
    fileBytes = None
    with open(inputFilePath, "rb") as f:
        fileBytes = f.read()
    return fileBytes

def readMemory(memAddr, byteNum, endian="little", signed=False):
    """read out value from memory"""
    global uc
    readoutRawValue = uc.mem_read(memAddr, byteNum)
    logging.info(" >> readoutRawValue hex=0x%s", readoutRawValue.hex())
    readoutValue = int.from_bytes(readoutRawValue, endian, signed=signed)
    logging.info(" >> readoutValue=0x%016X", readoutValue)
    return readoutValue

def writeMemory(memAddr, newValue, byteLen):
    """
        for ARM64 little endian, write new value into memory address
        memAddr: memory address to write
        newValue: value to write
        byteLen: 4 / 8
    """
    global uc

    valueFormat = "0x%016X" if byteLen == 8 else "0x%08X"
    if isinstance(newValue, bytes):
```

```
        logging.info("writeMemory: memAddr=0x%X, newValue=0x%s, byteLen=%d", memAddr, n
ewValue.hex(), byteLen)
        newValueBytes = newValue
    else:
        valueStr = valueFormat % newValue
        logging.info("writeMemory: memAddr=0x%X, newValue=%s, byteLen=%d", memAddr, val
ueStr, byteLen)
        newValueBytes = newValue.to_bytes(byteLen, "little")
    uc.mem_write(memAddr, newValueBytes)
    logging.info(" >> has write newValueBytes=%s to address=0x%X", newValueBytes, memAd
dr)

    # # for debug: verify write is OK or not
    # readoutValue = uc.mem_read(memAddr, byteLen)
    # logging.info("for address 0x%X, readoutValue hex=0x%s", memAddr, readoutValue.hex
()))
    # # logging.info("readoutValue hexlify=%b", binascii.hexlify(readoutValue))
    # readoutValueLong = int.from_bytes(readoutValue, "little", signed=False)
    # logging.info("readoutValueLong=0x%x", readoutValueLong)
    # # if readoutValue == newValue:
    # if readoutValueLong == newValue:
    #     logging.info("=== Write and read back OK")
    # else:
    #     logging.info("!!! Write and read back Failed")

def shouldStopEmulate(curPc, decodedInsn):
    isShouldStop = False
    # isRetInsn = decodedInsn.mnemonic == "ret"
    isRetInsn = re.match("^ret", decodedInsn.mnemonic) # support: ret/retaa/retab/...
    if isRetInsn:
        isPcInsideMainCode = (curPc >= CODE_ADDRESS) and (curPc < CODE_ADDRESS_REAL_END)

        isShouldStop = isRetInsn and isPcInsideMainCode

    return isShouldStop

# debug related

def bytesToOpcodeStr(curBytes):
    opcodeByteStr = ''.join('{:02X} '.format(eachByte) for eachByte in curBytes)
    return opcodeByteStr

def dbgAddressRangeStr(startAddress, size):
    endAddress = startAddress + (size - 1)
    addrRangeStr = "0x%X:0x%X" % (startAddress, endAddress)
    return addrRangeStr

################################################################################
# Main
################################################################################

# init logging
curLogFile = "%s_%s.log" % (getFilenameNoPointSuffix(__file__), getCurDatetimeStr())
# 'TIAutoOrder_20221201_174058.log'
curLogFullFile = os.path.join("debug", "log", curLogFile) # 'emulate_akd_getIDMSRouting
Info_20230529_094920.log'
```

```
# 'debug\\log\\TIAutoOrder_20221201_174112.log'
crifanLogging.loggingInit(filename=curLogFullFile)
# crifanLogging.testLogging()
# logging.debug("debug log")
# logging.info("info log")
logging.info("Output log to %s", curLogFullFile)


# Init Capstone instance
cs = Cs(CS_ARCH_ARM64, CS_MODE_ARM + CS_MODE_LITTLE_ENDIAN)
cs.detail = True

# Init Unicorn

# code to be emulated

# for arm64e: ___lldb_unnamed_symbol2540$$akd
# akd_symbol2540_FilePath = "input/akd_getIDMSRoutingInfo/arm64e/akd_arm64e_symbol2540.
bin"
# akd_symbol2540_FilePath = "input/akd_getIDMSRoutingInfo/arm64e/akd_arm64e_symbol2540_
noCanary.bin"
# akd_symbol2540_FilePath = "input/akd_getIDMSRoutingInfo/arm64e/akd_arm64e_symbol2540_
noCanary_braaToBr.bin"
# b"\x7F\x23\x03\xD5..."

# for arm64: ___lldb_unnamed_symbol2575$$akd
akd_symbol2575_FilePath = "input/akd_getIDMSRoutingInfo/arm64/akd_arm64_symbol2575.bin"
logging.info("akd_symbol2575_FilePath=%s", akd_symbol2575_FilePath)
ARM64_CODE_akd_symbol2575 = readBinFileBytes(akd_symbol2575_FilePath) # b'\xff\xc3\x03\
xd1\xfco\t\xa9\xfag\n\xa9\xf8_\x0b\xa9\xf6W\x0c\xa9\xf4O
gCodeSizeReal = len(ARM64_CODE_akd_symbol2575)
logging.info("gCodeSizeReal=%d == 0x%X", gCodeSizeReal, gCodeSizeReal)
# ___lldb_unnamed_symbol2540: 10064 == 0x2750
# ___lldb_unnamed_symbol2575 == sub_1000A0460: 9416 == 0x24C8

#------------------- Code --------------------

# memory address where emulation starts
CODE_ADDRESS = 0x10000
logging.info("CODE_ADDRESS=0x%X", CODE_ADDRESS)

# code size: 4MB
CODE_SIZE = 4 * 1024 * 1024
logging.info("CODE_SIZE=0x%X", CODE_SIZE)
CODE_ADDRESS_END = (CODE_ADDRESS + CODE_SIZE) # 0x00410000
logging.info("CODE_ADDRESS_END=0x%X", CODE_ADDRESS_END)

CODE_ADDRESS_REAL_END = CODE_ADDRESS + gCodeSizeReal
logging.info("CODE_ADDRESS_REAL_END=0x%X", CODE_ADDRESS_REAL_END)
# CODE_ADDRESS_REAL_LAST_LINE = CODE_ADDRESS_REAL_END - 4
# logging.info("CODE_ADDRESS_REAL_LAST_LINE=0x%X", CODE_ADDRESS_REAL_LAST_LINE)

#------------------- Try fix br jump UC_ERR_MAP -------------------

x9SmallOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_0x10
0d91680_0x100d938b0_x9SmallOffset.bin"
```

```
logging.info("x9SmallOffsetFile=%s", x9SmallOffsetFile)
x9SmallOffsetBytes = readBinFileBytes(x9SmallOffsetFile)
x9SmallOffsetBytesLen = len(x9SmallOffsetBytes) # b' \x00\x00\x00\xc0\x00\x00\x00\\\x00
\x00\x00D\x00\x00\x00h\x00\x00\x00H\x01 ...
# logging.info("x9SmallOffsetBytesLen=%d=0x%X", x9SmallOffsetBytesLen, x9SmallOffsetByt
esLen))

x9SmallOffsetStartAddress = CODE_ADDRESS + 0x21220
# logging.info("x9SmallOffsetStartAddress=0x%X", x9SmallOffsetStartAddress)
x9SmallOffsetEndAddress = x9SmallOffsetStartAddress + x9SmallOffsetBytesLen
# logging.info("x9SmallOffsetEndAddress=0x%X", x9SmallOffsetEndAddress)

# x10AbsFuncAddrWithOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_ar
m64_data_0x100dc8480_0x100dc9fe0_x10AbsFuncAddrWithOffset.bin"
x10AbsFuncAddrWithOffsetFile = "input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm6
4_data_x10EmulateAddr.bin"
logging.info("x10AbsFuncAddrWithOffsetFile=%s", x10AbsFuncAddrWithOffsetFile)
x10AbsFuncAddrWithOffsetBytes = readBinFileBytes(x10AbsFuncAddrWithOffsetFile)
# x10AbsFuncAddrWithOffsetBytesLen = len(x10AbsFuncAddrWithOffsetBytes) # b'\xa8F\xd6\x
00\x01\x00\x00\x00\x10G\xd6\x00\x01\x00\x00\x00lG\xd6\x00\x01 ...
x10AbsFuncAddrWithOffsetBytesLen = len(x10AbsFuncAddrWithOffsetBytes) # b'HB\x00\x00\x0
0\x00\x00\x00\xb0B\x00\x00\x00\x00\x00\x00\x0cC\x00\x00\x00\ ...
# logging.info("x10AbsFuncAddrWithOffsetBytesLen=%d=0x%X", x10AbsFuncAddrWithOffsetByte
sLen, x10AbsFuncAddrWithOffsetBytesLen)) # x10AbsFuncAddrWithOffsetBytesLen=7008=0x1B60

x10AbsFuncAddrWithOffsetStartAddress = CODE_ADDRESS + 0x58020
# logging.info("x10AbsFuncAddrWithOffsetStartAddress=0x%X", x10AbsFuncAddrWithOffsetSta
rtAddress)
x10AbsFuncAddrWithOffsetEndAddress = x10AbsFuncAddrWithOffsetStartAddress + x10AbsFuncA
ddrWithOffsetBytesLen
# logging.info("x10AbsFuncAddrWithOffsetEndAddress=0x%X", x10AbsFuncAddrWithOffsetEndAd
dress)

#------------------- emulate malloc --------------------
emulateMallocOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateMallocCodeSize = len(emulateMallocOpcode)

EMULATE_MALLOC_CODE_START = 2 * 1024 * 1024
EMULATE_MALLOC_CODE_END = EMULATE_MALLOC_CODE_START + gEmulateMallocCodeSize

MALLOC_JUMP_ADDR = 0x69BD8
MALLOC_JUMP_VALUE = EMULATE_MALLOC_CODE_START + 2
MALLOC_JUMP_SIZE = 8

#------------------- emulate free --------------------
emulateFreeOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateFreeCodeSize = len(emulateFreeOpcode)

EMULATE_FREE_CODE_START = (2 * 1024 * 1024) + (128 * 1024)
EMULATE_FREE_CODE_END = EMULATE_FREE_CODE_START + gEmulateFreeCodeSize

FREE_JUMP_ADDR = 0x69B88
FREE_JUMP_VALUE = EMULATE_FREE_CODE_START + 2
FREE_JUMP_SIZE = 8

#------------------- emulate demalloc --------------------
```

```
emulateDemallocOpcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateDemallocCodeSize = len(emulateDemallocOpcode)

EMULATE_DEMALLOC_CODE_START = (2 * 1024 * 1024) + (256 * 1024)
EMULATE_DEMALLOC_CODE_END = EMULATE_DEMALLOC_CODE_START + gEmulateDemallocCodeSize

DEMALLOC_JUMP_ADDR = 0x69C08
DEMALLOC_JUMP_VALUE = EMULATE_DEMALLOC_CODE_START + 2
DEMALLOC_JUMP_SIZE = 8

#------------------- emulate (call sub function) ___lldb_unnamed_symbol2567$$akd -----
--------------
emulateAkdFunc2567Opcode = b"\xC0\x03\x5F\xD6" # current only ret=0xC0035FD6
gEmulateAkdFunc2567Size = len(emulateAkdFunc2567Opcode)

EMULATE_AKD_FUNC_2567_START = (2 * 1024 * 1024) + (512 * 1024)
EMULATE_AKD_FUNC_2567_END = EMULATE_AKD_FUNC_2567_START + gEmulateAkdFunc2567Size

AKD_FUNC_2567_JUMP_ADDR = 0x69BC0
AKD_FUNC_2567_JUMP_VALUE = EMULATE_AKD_FUNC_2567_START + 3
AKD_FUNC_2567_JUMP_SIZE = 8

#------------------- misc jump address and value -------------------

LINE_7396_STORE_VALUE_ADDR = 0x80000

LINE_7392_JUMP_ADDR = 0x69BE8
LINE_7392_JUMP_VALUE = LINE_7396_STORE_VALUE_ADDR + 2
LINE_7392_JUMP_SIZE = 8


#------------------- __stack_chk_guard -------------------
# ->  0x10469c484 <+36>: ldr    x8, #0x54354              ; (void *)0x00000001f13db058:
 __stack_chk_guard
#       x8 = 0x00000001f13db058  libsystem_c.dylib`__stack_chk_guard
LIBC_ADDRESS = 5 * 1024 * 1024
LIBC_SIZE = 512 * 1024
STACK_CHECK_GUADR_ADDRESS = LIBC_ADDRESS + 0xB058

#------------------- Heap -------------------

HEAP_ADDRESS = 6 * 1024 * 1024
HEAP_SIZE = 1 * 1024 * 1024

HEAP_ADDRESS_END = HEAP_ADDRESS + HEAP_SIZE
HEAP_ADDRESS_LAST_BYTE = HEAP_ADDRESS_END - 1

#------------------- Stack -------------------
# Stack: from High address to lower address ?
STACK_ADDRESS = 7 * 1024 * 1024
STACK_SIZE = 1 * 1024 * 1024
STACK_HALF_SIZE = (int)(STACK_SIZE / 2)

# STACK_ADDRESS_END = STACK_ADDRESS - STACK_SIZE # 8 * 1024 * 1024
# STACK_SP = STACK_ADDRESS - 0x8 # ARM64: offset 0x8
```

```
# STACK_TOP = STACK_ADDRESS + STACK_SIZE
STACK_TOP = STACK_ADDRESS + STACK_HALF_SIZE
STACK_SP = STACK_TOP

FP_X29_VALUE = STACK_SP + 0x30

LR_INIT_ADDRESS = CODE_ADDRESS

#------------------- Args -------------------

# memory address for arguments
ARGS_ADDRESS = 8 * 1024 * 1024
ARGS_SIZE =  0x10000

# init args value
ARG_routingInfoPtr = ARGS_ADDRESS
ARG_DSID = 0xfffffffffffffffe

#------------------- Unicorn Hook -------------------

# callback for tracing basic blocks
def hook_block(mu, address, size, user_data):
    logging.info("@@@ Tracing basic block at 0x%x, block size = 0x%x", address, size)

# callback for tracing instructions
def hook_code(mu, address, size, user_data):
    global ucHeap

    pc = mu.reg_read(UC_ARM64_REG_PC)

    # logging.info(">>> Tracing instruction at 0x%x, instruction size = 0x%x", address,
 size)
    lineCount = int(size / BYTES_PER_LINE)
    for curLineIdx in range(lineCount):
        startAddress = address + curLineIdx * BYTES_PER_LINE
        codeOffset = startAddress - CODE_ADDRESS
        opcodeBytes = mu.mem_read(startAddress, BYTES_PER_LINE)
        opcodeByteStr = bytesToOpcodeStr(opcodeBytes)
        decodedInsnGenerator = cs.disasm(opcodeBytes, address)
        # if gSingleLineCode:
        for eachDecodedInsn in decodedInsnGenerator:
            eachInstructionName = eachDecodedInsn.mnemonic
            offsetStr = "<+%d>" % codeOffset
            logging.info("--- 0x%08X %7s: %s -> %s\t%s", startAddress, offsetStr, opcod
eByteStr, eachInstructionName, eachDecodedInsn.op_str)
            if shouldStopEmulate(pc, eachDecodedInsn):
                mu.emu_stop()
                logging.info("Emulate done!")

            gNoUse = 1

    # for debug
    toLogDict = {
        0x00010070: ["x25"],
        0x00010074: ["cpsr", "w9", "x9", "x25"],
```

```
        0x00010078: ["cpsr", "x9"],
        0x00010080: ["cpsr", "x9", "x10"],
        0x00010084: ["cpsr", "x9"],
        0x00010100: ["x24", "w10"],
        0x00010104: ["x10"],
        0x00010108: ["x27"],
        0x00200000: ["cpsr", "x0", "x1"],
        0x000100D0: ["x0", "x1"],
        0x000100F8: ["x9", "x10"],
        0x000100FC: ["x9"],
        0x0001011C: ["x9"],
        0x0001016C: ["w8", "x25"],
        0x00010170: ["x8"],
        0x00010178: ["x10"],
        0x00011124: ["w24"],
        0x00011128: ["w8"],
        0x0001112C: ["x9"],
        0x00011150: ["x8", "x9"],
        0x00011160: ["x0", "x1", "x2", "x3", "x4", "x26"],
        0x00011164: ["x0"],
        0x000118B4: ["x0", "x22"],
        0x000118B8: ["x0", "x9"],
        0x00011CE0: ["w8", "x9"],
        0x00011CE4: ["x8"],
        0x00011CFC: ["w8", "x9"],
        0x00011D00: ["x8"],
        0x00012138: ["sp"],
        0x00012430: ["x25", "w8"],
        0x00012434: ["x8"],
        0x0001243C: ["x8", "x9"],
        0x00012440: ["x8"],
        0x0001244C: ["x16"],
        0x00012450: ["x27"],
    }

    # common debug

    cpsr = mu.reg_read(UC_ARM_REG_CPSR)
    sp = mu.reg_read(UC_ARM_REG_SP)

    w8 = mu.reg_read(UC_ARM64_REG_W8)
    w9 = mu.reg_read(UC_ARM64_REG_W9)
    w10 = mu.reg_read(UC_ARM64_REG_W10)
    w11 = mu.reg_read(UC_ARM64_REG_W11)
    w24 = mu.reg_read(UC_ARM64_REG_W24)
    w26 = mu.reg_read(UC_ARM64_REG_W26)

    x0 = mu.reg_read(UC_ARM64_REG_X0)
    x1 = mu.reg_read(UC_ARM64_REG_X1)
    x2 = mu.reg_read(UC_ARM64_REG_X2)
    x3 = mu.reg_read(UC_ARM64_REG_X3)
    x4 = mu.reg_read(UC_ARM64_REG_X4)
    x8 = mu.reg_read(UC_ARM64_REG_X8)
    x9 = mu.reg_read(UC_ARM64_REG_X9)
    x10 = mu.reg_read(UC_ARM64_REG_X10)
    x16 = mu.reg_read(UC_ARM64_REG_X16)
```

```
    x22 = mu.reg_read(UC_ARM64_REG_X22)
    x24 = mu.reg_read(UC_ARM64_REG_X24)
    x25 = mu.reg_read(UC_ARM64_REG_X25)
    x26 = mu.reg_read(UC_ARM64_REG_X26)
    x27 = mu.reg_read(UC_ARM64_REG_X27)

    regNameToValueDict = {
        "cpsr": cpsr,
        "sp": sp,

        "w8": w8,
        "w9": w9,
        "w10": w10,
        "w11": w11,
        "w24": w24,
        "w26": w26,

        "x0": x0,
        "x1": x1,
        "x2": x2,
        "x3": x3,
        "x4": x4,
        "x8": x8,
        "x9": x9,
        "x10": x10,
        "x16": x16,
        "x22": x22,
        "x24": x24,
        "x25": x25,
        "x26": x26,
        "x27": x27,
    }

    toLogAddressList = toLogDict.keys()
    if pc in toLogAddressList:
        toLogRegList = toLogDict[pc]
        initLogStr = "\tdebug: PC=0x%X: " % pc
        regLogStrList = []
        for eachRegName in toLogRegList:
            eachReg = regNameToValueDict[eachRegName]
            isWordReg = re.match("x\d+", eachRegName)
            logFormt = "0x%016X" if isWordReg else "0x%08X"
            curRegValueStr = logFormt % eachReg
            curRegLogStr = "%s=%s" % (eachRegName, curRegValueStr)
            regLogStrList.append(curRegLogStr)
        allRegStr = ", ".join(regLogStrList)
        wholeLogStr = initLogStr + allRegStr
        logging.info("%s", wholeLogStr)
        gNoUse = 1

    # for emulateMalloc
    # if pc == 0x00200000:
    if pc == EMULATE_MALLOC_CODE_START:
        mallocSize = mu.reg_read(UC_ARM64_REG_X0)
        newAddrPtr = ucHeap.malloc(mallocSize)
        mu.reg_write(UC_ARM64_REG_X0, newAddrPtr)
```

```
        logging.info("\temulateMalloc: input x0=0x%x, output ret: 0x%x", mallocSize, ne
wAddrPtr)
        gNoUse = 1

    if pc == EMULATE_FREE_CODE_START:
        address = mu.reg_read(UC_ARM64_REG_X0)
        logging.info("\temulateFree: input address=0x%x", address)
        gNoUse = 1

    if pc == EMULATE_DEMALLOC_CODE_START:
        targetTask = mu.reg_read(UC_ARM64_REG_X0)
        address = mu.reg_read(UC_ARM64_REG_X1)
        size = mu.reg_read(UC_ARM64_REG_X2)
        # zeroValue = 0
        # zeroValueBytes = zeroValue.to_bytes(size, "little")
        if (address > 0) and (size > 0):
            writeMemory(address, 0, size)
        logging.info("\temulateDemalloc: input targetTask=0x%X,address=0x%X,size=%d=0x%
X", targetTask, address, size, size)
        gNoUse = 1

    if pc == EMULATE_AKD_FUNC_2567_START:
        paraX0 = mu.reg_read(UC_ARM64_REG_X0)
        paraX1 = mu.reg_read(UC_ARM64_REG_X1)
        paraX2 = mu.reg_read(UC_ARM64_REG_X2)
        paraX3 = mu.reg_read(UC_ARM64_REG_X3)
        paraX4 = mu.reg_read(UC_ARM64_REG_X4)

        realDebuggedRetValue = 0
        mu.reg_write(UC_ARM64_REG_X0, realDebuggedRetValue)
        logging.info("\temulateAkdFunc2567: input x0=0x%x,x1=0x%x,x2=0x%x,x3=0x%x,x4=0x
%x, output ret: 0x%x", paraX0,paraX1,paraX2,paraX3,paraX4, realDebuggedRetValue)
        gNoUse = 1

    # if pc == 0x00011754:
    #     logging.info("")

    # if pc == 0x0001010C:
    #     logging.info("")

    if pc == 0x12138:
        spValue = mu.mem_read(sp)
        logging.info("\tspValue=0x%X", spValue)
        gNoUse = 1

    if pc == 0x1213C:
        gNoUse = 1

    if pc == 0x118B4:
        gNoUse = 1

    if pc == 0x118B8:
        gNoUse = 1


def hook_unmapped(mu, access, address, size, value, context):
```

```
    pc = mu.reg_read(UC_ARM64_REG_PC)
    logging.info("!!! Memory UNMAPPED at 0x%X size=0x%x, access(r/w)=%d, value=0x%X, PC
=0x%X", address, size, access, value, pc)
    mu.emu_stop()
    return True

def hook_mem_write(uc, access, address, size, value, user_data):
    if address == ARG_routingInfoPtr:
        logging.info("write ARG_routingInfoPtr")
        gNoUse = 1

    pc = uc.reg_read(UC_ARM64_REG_PC)
    logging.info(" >> Memory WRITE at 0x%X, size=%u, value=0x%X, PC=0x%X", address, size
, value, pc)
    # logging.info(" >> Memory WRITE at 0x%X, size=%u, value=0x%s, PC=0x%X", address, s
ize, value.to_bytes(8, "little").hex(), pc))
    gNoUse = 1

def hook_mem_read(uc, access, address, size, value, user_data):
    if address == ARG_routingInfoPtr:
        logging.info("read ARG_routingInfoPtr")
        gNoUse = 1

    pc = uc.reg_read(UC_ARM64_REG_PC)
    data = uc.mem_read(address, size)
    logging.info(" << Memory READ at 0x%X, size=%u, rawValueLittleEndian=0x%s, pc=0x%X",
 address, size, data.hex(), pc)
    gNoUse = 1

    dataLong = int.from_bytes(data, "little", signed=False)
    if dataLong == 0:
        logging.info(" !! Memory read out 0 -> possbile abnormal -> need attention")
        gNoUse = 1


# def hook_mem_fetch(uc, access, address, size, value, user_data):
#     pc = uc.reg_read(UC_ARM64_REG_PC)
#     logging.info(" >> Memory FETCH at 0x%X, size= %u, value= 0x%X, PC= 0x%X", address
, size, value, pc))
#     gNoUse = 1

#------------------- Unicorn main --------------------

# Emulate arm function running
def emulate_akd_arm64_symbol2575():
    global uc, ucHeap
    logging.info("Emulate arm64 sub_1000A0460 == ___lldb_unnamed_symbol2575$$akd functi
on running")
    try:
        # Initialize emulator in ARM mode
        # mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)
        mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM + UC_MODE_LITTLE_ENDIAN)
        uc = mu
        # map code memory for this emulation
        mu.mem_map(CODE_ADDRESS, CODE_SIZE)
        logging.info("Mapped memory: Code\t[0x%08X-0x%08X]", CODE_ADDRESS, CODE_ADDRESS
```

```
+ CODE_SIZE)
        # code sub area
        logging.info("\t\t\t  [0x%08X-0x%08X] func: ___lldb_unnamed_symbol2575$$akd", C
ODE_ADDRESS, CODE_ADDRESS_REAL_END)
        logging.info("\t\t\t  [0x%08X-0x%08X]   fix br err: x9SmallOffset", x9SmallOffs
etStartAddress, x9SmallOffsetEndAddress)
        logging.info("\t\t\t  [0x%08X-0x%08X]   fix br err: x10AbsFuncAddrWithOffset",
x10AbsFuncAddrWithOffsetStartAddress, x10AbsFuncAddrWithOffsetEndAddress)
        logging.info("\t\t\t  [0x%08X-0x%08X]   emulateFree jump", FREE_JUMP_ADDR, FREE
_JUMP_ADDR + FREE_JUMP_SIZE)
        logging.info("\t\t\t  [0x%08X-0x%08X]   emulateAkdFunc2567 jump", AKD_FUNC_2567
_JUMP_ADDR, AKD_FUNC_2567_JUMP_ADDR + AKD_FUNC_2567_JUMP_SIZE)
        logging.info("\t\t\t  [0x%08X-0x%08X]   emulateMalloc jump", MALLOC_JUMP_ADDR,
MALLOC_JUMP_ADDR + MALLOC_JUMP_SIZE)
        logging.info("\t\t\t  [0x%08X-0x%08X]   line 7392 jump", LINE_7392_JUMP_ADDR, L
INE_7392_JUMP_ADDR + LINE_7392_JUMP_SIZE)
        logging.info("\t\t\t  [0x%08X-0x%08X]   emulateDemalloc jump", DEMALLOC_JUMP_AD
DR, DEMALLOC_JUMP_ADDR + DEMALLOC_JUMP_SIZE)
        logging.info("\t\t\t  [0x%08X-0x%08X] func: emulateMalloc", EMULATE_MALLOC_CODE
_START, EMULATE_MALLOC_CODE_END)
        logging.info("\t\t\t  [0x%08X-0x%08X] func: emulateFree", EMULATE_FREE_CODE_STA
RT, EMULATE_FREE_CODE_END)
        logging.info("\t\t\t  [0x%08X-0x%08X] func: emulateAkdFunc2567", EMULATE_AKD_FU
NC_2567_START, EMULATE_AKD_FUNC_2567_END)

        # map libc, for __stack_chk_guard
        mu.mem_map(LIBC_ADDRESS, LIBC_SIZE)
        logging.info("Mapped memory: Libc\t[0x%08X-0x%08X]", LIBC_ADDRESS, LIBC_ADDRESS
+ LIBC_SIZE)
        # map heap
        mu.mem_map(HEAP_ADDRESS, HEAP_SIZE)
        logging.info("Mapped memory: Heap\t[0x%08X-0x%08X]", HEAP_ADDRESS, HEAP_ADDRESS
+ HEAP_SIZE)
        # map stack
        mu.mem_map(STACK_ADDRESS, STACK_SIZE)
        # mu.mem_map(STACK_ADDRESS_END, STACK_SIZE)
        logging.info("Mapped memory: Stack\t[0x%08X-0x%08X]", STACK_ADDRESS, STACK_ADDR
ESS + STACK_SIZE)
        # map arguments
        mu.mem_map(ARGS_ADDRESS, ARGS_SIZE)
        logging.info("Mapped memory: Args\t[0x%08X-0x%08X]", ARGS_ADDRESS, ARGS_ADDRESS
+ ARGS_SIZE)

        # init Heap malloc emulation
        ucHeap = UnicornSimpleHeap(uc, HEAP_ADDRESS, HEAP_ADDRESS_LAST_BYTE, debug_print
=True)

        # write machine code to be emulated to memory
        # mu.mem_write(CODE_ADDRESS, ARM64_CODE_akd_symbol2540)
        mu.mem_write(CODE_ADDRESS, ARM64_CODE_akd_symbol2575)

        # # for debug: test memory set to 0
        # testAddr = 0x300000
        # testInt = 0x12345678
        # testIntBytes = testInt.to_bytes(8, "little", signed=False)
        # mu.mem_write(testAddr, testIntBytes)
```

```
        # readoutInt1 = readMemory(testAddr, 8)
        # logging.info("readoutInt1=0x%x", readoutInt1)
        # writeMemory(testAddr, 0, 3)
        # readoutInt2 = readMemory(testAddr, 8)
        # logging.info("readoutInt2=0x%x", readoutInt2)

        mu.mem_write(x9SmallOffsetStartAddress, x9SmallOffsetBytes)
        logging.info(" >> has write %d=0x%X bytes into memory [0x%X-0x%X]", x9SmallOffs
etBytesLen, x9SmallOffsetBytesLen, x9SmallOffsetStartAddress, x9SmallOffsetStartAddress
+ x9SmallOffsetBytesLen)
        mu.mem_write(x10AbsFuncAddrWithOffsetStartAddress, x10AbsFuncAddrWithOffsetBytes
)
        logging.info(" >> has write %d=0x%X bytes into memory [0x%X-0x%X]", x10AbsFuncA
ddrWithOffsetBytesLen, x10AbsFuncAddrWithOffsetBytesLen, x10AbsFuncAddrWithOffsetStartA
ddress, x10AbsFuncAddrWithOffsetStartAddress + x10AbsFuncAddrWithOffsetBytesLen)

        # for emuleateMalloc
        writeMemory(EMULATE_MALLOC_CODE_START, emulateMallocOpcode, gEmulateMallocCodeS
ize)
        # writeMemory(0x69BD8, EMULATE_MALLOC_CODE_START + 2, 8)
        writeMemory(MALLOC_JUMP_ADDR, MALLOC_JUMP_VALUE, MALLOC_JUMP_SIZE)

        # for emuleateFree
        writeMemory(EMULATE_FREE_CODE_START, emulateFreeOpcode, gEmulateFreeCodeSize)
        writeMemory(FREE_JUMP_ADDR, FREE_JUMP_VALUE, FREE_JUMP_SIZE) # <+256>: 0A DB 6A
 F8  -> ldr     x10, [x24, w10, sxtw #3]

        # for emuleateDemalloc
        writeMemory(EMULATE_DEMALLOC_CODE_START, emulateDemallocOpcode, gEmulateDemallo
cCodeSize)
        writeMemory(DEMALLOC_JUMP_ADDR, DEMALLOC_JUMP_VALUE, DEMALLOC_JUMP_SIZE) # <+74
20>: 28 D9 68 F8  -> ldr     x8, [x9, w8, sxtw #3]

        # for emulateAkdFunc2567
        writeMemory(EMULATE_AKD_FUNC_2567_START, emulateAkdFunc2567Opcode, gEmulateAkdF
unc2567Size)
        # writeMemory(0x69BC0, EMULATE_AKD_FUNC_2567_START + 3, 8) # <+4432>: 28 D9 68
F8  -> ldr     x8, [x9, w8, sxtw #3]
        writeMemory(AKD_FUNC_2567_JUMP_ADDR, AKD_FUNC_2567_JUMP_VALUE, AKD_FUNC_2567_JU
MP_SIZE) # <+4432>: 28 D9 68 F8  -> ldr     x8, [x9, w8, sxtw #3]

        # initialize some memory

        # for arm64e:
        # writeMemory(0x757DC, 0x0000000100af47c2, 8)
        # writeMemory(0x662FC, 0x237d5780000100A0, 8)

        # for arm64:

        # for __stack_chk_guard
        writeMemory(0x64378, STACK_CHECK_GUADR_ADDRESS, 4)
        writeMemory(0x50B058, 0x75c022d064c70008, 8)

        # Note: following addr and value have been replaced by: x9 and x10, two group a
ddr and values
        # writeMemory(0x32850, 0x00000094, 4)             # <+236>: 29 DB A9 B8  -> ldr
```

```
sw   x9, [x25, w9, sxtw #2]
        # readMemory(0x32850, 4)
        # writeMemory(0x32870, 0xffffdbc4, 4)     # <+116>: 29 DB A9 B8  -> ldrsw   x9,
 [x25, w9, sxtw #2]
        # readMemory(0x32870, 4)
        # writeMemory(0x68CF8, CODE_ADDRESS_REAL_END, 8)  # <+124>: EA 63 2C 58  -> ldr
     x10, #0x68cf8
        # readMemory(0x68CF8, 8)
        # writeMemory(0x68D00, 0x1008C, 8)        # <+244>: 6A 60 2C 58  -> ldr     x10
, #0x68d00
        # readMemory(0x68D00, 8)
        # writeMemory(0x32858, 0xc4, 4)           # <+364>: 28 DB A8 B8  -> ldrsw   x8,
 [x25, w8, sxtw #2]
        # readMemory(0x32858, 4)
        # writeMemory(0x68D08, 0x10120, 8)        # <+372>: AA 5C 2C 58  -> ldr     x10
, #0x68d08
        # readMemory(0x68D08, 8)

        writeMemory(0x69C18, 0x0000000000078dfa, 8) # <+4400>: 36 D9 68 F8  -> ldr
x22, [x9, w8, sxtw #3]
        writeMemory(0x78DF8, 0x0000000000003f07, 8) # <+4404>: C0 EE 5F B8  -> ldr
w0, [x22, #-2]!

        writeMemory(LINE_7392_JUMP_ADDR, LINE_7392_JUMP_VALUE, LINE_7392_JUMP_SIZE) # <
+7392>: 28 D9 68 F8  -> ldr     x8, [x9, w8, sxtw #3]
        writeMemory(LINE_7396_STORE_VALUE_ADDR, 0x00000203, 4) # <+7396>: 00 E1 5F B8
-> ldur    w0, [x8, #-2]

        # initialize machine registers

        # # for arm64e arm64e ___lldb_unnamed_symbol2540$$akd
        # mu.reg_write(UC_ARM64_REG_X0, ARG_routingInfoPtr)
        # mu.reg_write(UC_ARM64_REG_X1, ARG_DSID)

        # for current arm64 ___lldb_unnamed_symbol2575$$akd =====
        mu.reg_write(UC_ARM64_REG_X0, ARG_DSID)
        mu.reg_write(UC_ARM64_REG_X1, ARG_routingInfoPtr)

        # mu.reg_write(UC_ARM64_REG_LR, CODE_ADDRESS_END)
        mu.reg_write(UC_ARM64_REG_LR, LR_INIT_ADDRESS)

        # initialize stack
        # mu.reg_write(UC_ARM64_REG_SP, STACK_ADDRESS)
        mu.reg_write(UC_ARM64_REG_SP, STACK_SP)

        mu.reg_write(UC_ARM64_REG_FP, FP_X29_VALUE)

        # tracing all basic blocks with customized callback
        mu.hook_add(UC_HOOK_BLOCK, hook_block)

        # tracing one instruction with customized callback
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS)
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_REA
L_END)
        # mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=EMULATE_MALLOC_C
ODE_END)
```

```
        mu.hook_add(UC_HOOK_CODE, hook_code, begin=CODE_ADDRESS, end=CODE_ADDRESS_END)

        # hook unmamapped memory
        mu.hook_add(UC_HOOK_MEM_UNMAPPED, hook_unmapped)

        # hook memory read and write
        mu.hook_add(UC_HOOK_MEM_READ, hook_mem_read)
        mu.hook_add(UC_HOOK_MEM_WRITE, hook_mem_write)
        # mu.hook_add(UC_HOOK_MEM_FETCH, hook_mem_fetch)

        logging.info("---------- Emulation Start ----------")

        # emulate machine code in infinite time
        mu.emu_start(CODE_ADDRESS, CODE_ADDRESS + len(ARM64_CODE_akd_symbol2575))

        # now print out some registers
        logging.info("---------- Emulation done. Below is the CPU context ----------")

        retVal = mu.reg_read(UC_ARM64_REG_X0)
        # routingInfo = mu.mem_read(ARG_routingInfoPtr)
        # logging.info(">>> retVal=0x%x, routingInfo=%d", retVal, routingInfo))
        logging.info(">>> retVal=0x%x", retVal)

        routingInfoEnd = mu.mem_read(ARG_routingInfoPtr, 8)
        logging.info(">>> routingInfoEnd hex=0x%s", routingInfoEnd.hex())
        routingInfoEndLong = int.from_bytes(routingInfoEnd, "little", signed=False)
        logging.info(">>> routingInfoEndLong=%d", routingInfoEndLong)

    except UcError as e:
        logging.info("ERROR: %s", e)
        logging.info("\n")

if __name__ == '__main__':
    emulate_akd_arm64_symbol2575()
    logging.info("=" * 26)
```

输出log

```
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:157  INFO    Output log to debug/log/emulate_akd_getIDMSRoutingInfo_20230607_230151.log
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:176  INFO    akd_symbol2575_FilePath: input/akd_getIDMSRoutingInfo/arm64/akd_arm64_symbol2575.bin
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:179  INFO    gCodeSizeReal=9416 == 0x24C8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:187  INFO    CODE_ADDRESS=0x10000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:191  INFO    CODE_SIZE=0x400000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:193  INFO    CODE_ADDRESS_END=0x410000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:196  INFO    CODE_ADDRESS_REAL_END=0x124C8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:203  INFO    x9SmallOffsetFile=input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_0x100d91680_0x100d938b0_x9SmallOffset.bin
```

```
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:215  INFO    x10AbsFuncAddrWithOffs
etFile input/akd_getIDMSRoutingInfo/arm64/lldb_memory/akd_arm64_data_x10EmulateAddr.bin
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:563  INFO    Emulate arm64 sub_1000
A0460 == ___lldb_unnamed_symbol2575$$akd function running
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:571  INFO    Mapped memory: Code
[0x00010000-0x00410000]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:573  INFO
   [0x00010000-0x000124C8] func: ___lldb_unnamed_symbol2575$$akd
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:574  INFO
   [0x00031220-0x00033450]   fix br err: x9SmallOffset
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:575  INFO
   [0x00068020-0x00069B80]   fix br err: x10AbsFuncAddrWithOffset
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:576  INFO
   [0x00069B88-0x00069B90]   emulateFree jump
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:577  INFO
   [0x00069BC0-0x00069BC8]   emulateAkdFunc2567 jump
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:578  INFO
   [0x00069BD8-0x00069BE0]   emulateMalloc jump
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:579  INFO
   [0x00069BE8-0x00069BF0]   line 7392 jump
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:580  INFO
   [0x00069C08-0x00069C10]   emulateDemalloc jump
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:581  INFO
   [0x00200000-0x00200004] func: emulateMalloc
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:582  INFO
   [0x00220000-0x00220004] func: emulateFree
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:583  INFO
   [0x00280000-0x00280004] func: emulateAkdFunc2567
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:587  INFO    Mapped memory: Libc
[0x00500000-0x00580000]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:590  INFO    Mapped memory: Heap
[0x00600000-0x00700000]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:594  INFO    Mapped memory: Stack
[0x00700000-0x00800000]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:597  INFO    Mapped memory: Args
[0x00800000-0x00810000]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:618  INFO     >> has write 8752 0x2
230 bytes into memory [0x31220-0x33450]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:620  INFO     >> has write 7008 0x1
B60 bytes into memory [0x68020-0x69B80]
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:102  INFO    writeMemory: memAddr 0
x200000, newValue 0xc0035fd6, byteLen 4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO     >> has write newValue
Bytes b'\xc0\x03_\xd6' to address 0x200000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO    writeMemory: memAddr 0
x69BD8, newValue 0x0000000000200002, byteLen 8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO     >> has write newValue
Bytes b'\x02\x00 \x00\x00\x00\x00\x00' to address 0x69BD8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:102  INFO    writeMemory: memAddr 0
x220000, newValue 0xc0035fd6, byteLen 4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO     >> has write newValue
Bytes b'\xc0\x03_\xd6' to address 0x220000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO    writeMemory: memAddr 0
x69B88, newValue 0x0000000000220002, byteLen 8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO     >> has write newValue
Bytes b'\x02\x00"\x00\x00\x00\x00\x00' to address 0x69B88
```

```
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:102  INFO     writeMemory: memAddr=0
x240000, newValue=0xc0035fd6, byteLen=4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\xc0\x03_\xd6' to address=0x240000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x69C08, newValue=0x0000000000240002, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x02\x00$\x00\x00\x00\x00\x00' to address=0x69C08
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:102  INFO     writeMemory: memAddr=0
x280000, newValue=0xc0035fd6, byteLen=4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\xc0\x03_\xd6' to address=0x280000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x69BC0, newValue=0x0000000000280003, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x03\x00(\x00\x00\x00\x00\x00' to address=0x69BC0
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x64378, newValue=0x0050B058, byteLen=4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'X\xb0P\x00' to address=0x64378
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x50B058, newValue=0x75C022D064C70008, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x08\x00\xc7d\xd0"\xc0u' to address=0x50B058
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x69C18, newValue=0x0000000000078DFA, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\xfa\x8d\x07\x00\x00\x00\x00\x00' to address=0x69C18
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x78DF8, newValue=0x0000000000003F07, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x07?\x00\x00\x00\x00\x00\x00' to address=0x78DF8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x69BE8, newValue=0x0000000000080002, byteLen=8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x02\x00\x08\x00\x00\x00\x00\x00' to address=0x69BE8
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:106  INFO     writeMemory: memAddr=0
x80000, newValue=0x00000203, byteLen=4
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:109  INFO      >> has write newValue
Bytes=b'\x03\x02\x00\x00' to address=0x80000
20230607 23:01:51 emulate_akd_getIDMSRoutingInfo.py:708  INFO     ---------- Emulation S
tart ----------
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO     @@@ Tracing basic bloc
k at 0x10000, block size = 0x8c
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010000     +0
: FF C3 03 D1  -> sub    sp, sp, #0xf0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010004     +4
: FC 6F 09 A9  -> stp    x28, x27, [sp, #0x90]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO      >> Memory WRITE at 0x
77FFA0, size=8, value=0x0, PC=0x10004
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO      >> Memory WRITE at 0x
77FFA8, size=8, value=0x0, PC=0x10004
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010008     +8
: FA 67 0A A9  -> stp    x26, x25, [sp, #0xa0]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO      >> Memory WRITE at 0x
77FFB0, size=8, value=0x0, PC=0x10008
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFB8, size 8, value 0x0, PC 0x10008
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001000C   +12
: F8 5F 0B A9  -> stp    x24, x23, [sp, #0xb0]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFC0, size 8, value 0x0, PC 0x1000C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFC8, size 8, value 0x0, PC 0x1000C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010010   +16
: F6 57 0C A9  -> stp    x22, x21, [sp, #0xc0]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFD0, size 8, value 0x0, PC 0x10010
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFD8, size 8, value 0x0, PC 0x10010
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010014   +20
: F4 4F 0D A9  -> stp    x20, x19, [sp, #0xd0]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFE0, size 8, value 0x0, PC 0x10014
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFE8, size 8, value 0x0, PC 0x10014
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010018   +24
: FD 7B 0E A9  -> stp    x29, x30, [sp, #0xe0]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFF0, size 8, value 0x780030, PC 0x10018
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FFF8, size 8, value 0x10000, PC 0x10018
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001001C   +28
: FD 83 03 91  -> add    x29, sp, #0xe0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010020   +32
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010024   +36
: A8 1A 2A 58  -> ldr    x8, #0x64378
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:544  INFO     << Memory READ at 0x6
4378, size 8, rawValueLittleEndian 0x58b0500000000000, pc 0x10024
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010028   +40
: 08 01 40 F9  -> ldr    x8, [x8]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:544  INFO     << Memory READ at 0x5
0B058, size 8, rawValueLittleEndian 0x0800c764d022c075, pc 0x10028
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001002C   +44
: A8 83 1A F8  -> stur   x8, [x29, #-0x58]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FF98, size 8, value 0x75C022D064C70008, PC 0x1002C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010030   +48
: FA 50 8B 52  -> movz   w26, #0x5a87
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010034   +52
: 9A 84 AD 72  -> movk   w26, #0x6c24, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010038   +56
: 08 18 00 91  -> add    x8, x0, #6
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001003C   +60
: 1F 15 00 F1  -> cmp    x8, #5
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010040   +64
: 04 28 48 BA  -> ccmn   x0, #8, #4, hs
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010044   +68
: 28 00 80 52  -> movz   w8, #0x1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010048   +72
: E8 03 88 1A  -> csel   w8, wzr, w8, eq
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001004C   +76
: 4B B4 94 52  -> movz   w11, #0xa5a2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010050   +80
: 6B 7B B2 72  -> movk   w11, #0x93db, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010054   +84
: 3F 00 00 F1  -> cmp    x1, #0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010058   +88
: E9 17 9F 1A  -> cset   w9, eq
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001005C   +92
: 28 01 08 2A  -> orr    w8, w9, w8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010060   +96
: 49 03 08 0B  -> add    w9, w26, w8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010064   +100
: 29 01 0B 0B  -> add    w9, w9, w11
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010068   +104
: 29 85 00 51  -> sub    w9, w9, #0x21
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001006C   +108
: 39 3F 11 10  -> adr    x25, #0x32850
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010070   +112
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
70: x25 0x0000000000032850
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010074   +116
: 29 DB A9 B8  -> ldrsw  x9, [x25, w9, sxtw #2]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
74: cpsr 0x20000000, w9 0x00000008, x9 0x0000000000000008, x25 0x0000000000032850
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:544  INFO      << Memory READ at 0x3
2870, size 4, rawValueLittleEndian 0xc4dbffff, pc 0x10074
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010078   +120
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
78: cpsr 0x20000000, x9 0xFFFFFFFFFFFFDBC4
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001007C   +124
: EA 63 2C 58  -> ldr    x10, #0x68cf8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:544  INFO      << Memory READ at 0x6
8CF8, size 8, rawValueLittleEndian 0xc824010000000000, pc 0x1007C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010080   +128
: 29 01 0A 8B  -> add    x9, x9, x10
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
80: cpsr 0x20000000, x9 0xFFFFFFFFFFFFDBC4, x10 0x00000000000124C8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010084   +132
: 16 F9 95 12  -> movn   w22, #0xafc8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
84: cpsr 0x20000000, x9 0x000000000001008C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010088   +136
: 20 01 1F D6  -> br     x9
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO     @@@ Tracing basic bloc
k at 0x1008c, block size = 0x44
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001008C   +140
: F7 03 01 AA  -> mov    x23, x1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010090   +144
: FC 03 00 AA  -> mov    x28, x0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010094   +148
: 08 01 00 52  -> eor    w8, w8, #1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010098   +152
: 69 A5 00 51  -> sub    w9, w11, #0x29
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001009C  +156
: 16 69 09 1B  -> madd   w22, w8, w9, w26
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100A0  +160
: 14 26 95 D2  -> movz   x20, #0xa930
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100A4  +164
: 34 4B BD F2  -> movk   x20, #0xea59, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100A8  +168
: B4 7B D3 F2  -> movk   x20, #0x9bdd, lsl #32
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100AC  +172
: 14 AE FA F2  -> movk   x20, #0xd570, lsl #48
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100B0  +176
: C8 2E 00 11  -> add    w8, w22, #0xb
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100B4  +180
: 78 D6 2C 10  -> adr    x24, #0x69b80
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100B8  +184
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100BC  +188
: 08 DB 68 F8  -> ldr    x8, [x24, w8, sxtw #3]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:544  INFO      << Memory READ at 0x6
9BD8, size 8, rawValueLittleEndian 0x0200200000000000, pc 0x100BC
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100C0  +192
: 08 09 00 D1  -> sub    x8, x8, #2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100C4  +196
: 00 03 80 52  -> movz   w0, #0x18
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100C8  +200
: E8 33 00 F9  -> str    x8, [sp, #0x60]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO      >> Memory WRITE at 0x
77FF70, size 8, value 0x200000, PC 0x100C8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100CC  +204
: 00 01 3F D6  -> blr    x8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO     @@@ Tracing basic bloc
k at 0x200000, block size = 0x4
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00200000  +203161
6: C0 03 5F D6  -> ret
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x200
000: cpsr 0x20000000, x0 0x0000000000000018, x1 0x0000000000800000
20230607 23:01:56 UnicornSimpleHeap.py:120  INFO     ++ Allocated heap chunk: chunk: [0
x600000-0x603000] ptr 0x601000, size 24 0x18
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:469  INFO           emulateMalloc:
input x0 0x18, output ret: 0x601000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO     @@@ Tracing basic bloc
k at 0x100d0, block size = 0x50
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100D0  +208
: 08 00 80 52  -> movz   w8, #0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
D0: x0 0x0000000000601000, x1 0x0000000000800000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100D4  +212
: 1F 00 00 F1  -> cmp    x0, #0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100D8  +216
: E9 07 9F 1A  -> cset   w9, ne
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100DC  +220
: EA 17 9F 1A  -> cset   w10, eq
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100E0  +224
: C9 06 09 0B  -> add    w9, w22, w9, lsl #1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100E4  +228
: 53 25 1A 1B  -> madd   w19, w10, w26, w9
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100E8  +232
: C9 16 96 1A  -  cinc   w9, w22, eq
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100EC  +236
: 29 DB A9 B8  -  ldrsw  x9, [x25, w9, sxtw #2]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100F0  +240
: 1F 20 03 D5  -  nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100F4  +244
: 6A 60 2C 58  -  ldr    x10, #0x68d00
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100F8  +248
: 29 01 0A 8B  -  add    x9, x9, x10
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
F8: x9 0x0000000000000094, x10 0x000000000001008C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x000100FC  +252
: CA 06 00 11  -  add    w10, w22, #1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x100
FC: x9 0x0000000000010120
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010100  +256
: 0A DB 6A F8  -  ldr    x10, [x24, w10, sxtw #3]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x101
00: x24 0x0000000000069B80, w10 0x00000001
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010104  +260
: 5B 09 00 D1  -  sub    x27, x10, #2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x101
04: x10 0x0000000000220002
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010108  +264
: 56 F9 95 12  -  movn   w22, #0xafca
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x101
08: x27 0x0000000000220000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001010C  +268
: 10 26 95 D2  -  movz   x16, #0xa930
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010110  +272
: 30 4B BD F2  -  movk   x16, #0xea59, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010114  +276
: B0 7B D3 F2  -  movk   x16, #0x9bdd, lsl #32
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010118  +280
: 10 AE FA F2  -  movk   x16, #0xd570, lsl #48
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001011C  +284
: 20 01 1F D6  -  br     x9
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO           debug: PC 0x101
1C: x9 0x0000000000010120
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO     @@@ Tracing basic bloc
k at 0x10120, block size = 0x2c
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010120  +288
: F5 03 00 AA  -  mov    x21, x0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010124  +292
: 14 00 14 8B  -  add    x20, x0, x20
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010128  +296
: 08 F3 8B D2  -  movz   x8, #0x5f98
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001012C  +300
: 68 AB A4 F2  -  movk   x8, #0x255b, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010130  +304
: 08 B9 D1 F2  -  movk   x8, #0x8dc8, lsl #32
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010134  +308
: 68 3E E7 F2  -  movk   x8, #0x39f3, lsl #48
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010138  +312
: 1F 20 00 A9  -  stp    xzr, x8, [x0]
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
601000, size 8, value 0x0, PC 0x10138
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
601008, size 8, value 0x39F38DC8255B5F98, PC 0x10138
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001013C  +316
: 1F 10 00 B9  -> str    wzr, [x0, #0x10]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
601010, size 4, value 0x0, PC 0x1013C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010140  +320
: 00 00 82 52  -> movz   w0, #0x1000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010144  +324
: E8 33 40 F9  -> ldr    x8, [sp, #0x60]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010148  +328
: 00 01 3F D6  -> blr    x8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO    @@@ Tracing basic bloc
k at 0x200000, block size = 0x4
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00200000  +203161
6: C0 03 5F D6  -> ret
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO          debug: PC 0x200
000: cpsr 0x20000000, x0 0x0000000000001000, x1 0x0000000000800000
20230607 23:01:56 UnicornSimpleHeap.py:120  INFO     ++ Allocated heap chunk: chunk: [0
x603000-0x606000] ptr 0x604000, size 4096 0x1000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:469  INFO          emulateMalloc:
input x0 0x1000, output ret: 0x604000
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO    @@@ Tracing basic bloc
k at 0x1014c, block size = 0x34
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001014C  +332
: A0 02 00 F9  -> str    x0, [x21]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
601000, size 8, value 0x604000, PC 0x1014C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010150  +336
: 48 0B 00 51  -> sub    w8, w26, #2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010154  +340
: 1F 00 00 F1  -> cmp    x0, #0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010158  +344
: E9 17 9F 1A  -> cset   w9, eq
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001015C  +348
: EA 07 9F 1A  -> cset   w10, ne
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010160  +352
: 48 4D 08 1B  -> madd   w8, w10, w8, w19
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010164  +356
: 09 09 09 0B  -> add    w9, w8, w9, lsl #2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010168  +360
: 68 16 93 1A  -> cinc   w8, w19, eq
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001016C  +364
: 28 DB A8 B8  -> ldrsw  x8, [x25, w8, sxtw #2]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO          debug: PC 0x101
6C: w8 0x00000002, x25 0x0000000000032850
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010170  +368
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO          debug: PC 0x101
70: x8 0x00000000000000C4
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010174  +372
: AA 5C 2C 58  -> ldr    x10, #0x68d08
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010178  +376
: 08 01 0A 8B  -> add    x8, x8, x10
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:460  INFO          debug: PC 0x101
78: x10 0x0000000000010120
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001017C   +380
: 00 01 1F D6  -> br     x8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO    @@@ Tracing basic bloc
k at 0x101e4, block size = 0x48
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101E4   +484
: 16 00 80 52  -> movz   w22, #0
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101E8   +488
: 08 F3 8B 52  -> movz   w8, #0x5f98
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101EC   +492
: 68 AB A4 72  -> movk   w8, #0x255b, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101F0   +496
: 08 05 40 11  -> add    w8, w8, #1, lsl #12
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101F4   +500
: A8 0A 00 B9  -> str    w8, [x21, #8]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
601008, size 4, value 0x255B6F98, PC 0x101F4
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101F8   +504
: 28 00 80 52  -> movz   w8, #0x1
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x000101FC   +508
: F3 03 09 AA  -> mov    x19, x9
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010200   +512
: F0 03 14 AA  -> mov    x16, x20
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010204   +516
: 55 B4 94 52  -> movz   w21, #0xa5a2
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010208   +520
: 75 7B B2 72  -> movk   w21, #0x93db, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001020C   +524
: A9 02 13 0B  -> add    w9, w21, w19
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010210   +528
: 29 01 08 0B  -> add    w9, w9, w8
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010214   +532
: 29 65 00 51  -> sub    w9, w9, #0x19
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010218   +536
: 29 DB A9 B8  -> ldrsw  x9, [x25, w9, sxtw #2]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001021C   +540
: 1F 20 03 D5  -> nop
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010220   +544
: CA 57 2C 58  -> ldr    x10, #0x68d18
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010224   +548
: 29 01 0A 8B  -> add    x9, x9, x10
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010228   +552
: 20 01 1F D6  -> br     x9
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:326  INFO    @@@ Tracing basic bloc
k at 0x1022c, block size = 0x84
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x0001022C   +556
: FC 13 00 F9  -> str    x28, [sp, #0x20]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FF30, size 8, value 0x-2, PC 0x1022C
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010230   +560
: F7 03 00 F9  -> str    x23, [sp]
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:533  INFO     >> Memory WRITE at 0x
77FF10, size 8, value 0x800000, PC 0x10230
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO    --- 0x00010234   +564
: 09 DA 8A D2  -> movz   x9, #0x56d0
```

```
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010238  +568
: C9 B4 A2 F2  -> movk   x9, #0x15a6, lsl #16
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x0001023C  +572
: 49 84 CC F2  -> movk   x9, #0x6422, lsl #32
20230607 23:01:56 emulate_akd_getIDMSRoutingInfo.py:346  INFO     --- 0x00010240  +576
: E9 51 E5 F2  -> movk   x9, #0x2a8f, lsl #48
...
```

辅助代码: `libs/UnicornSimpleHeap.py`

```
# Function: Emulate memory management (malloc/free/...)
# Author: Crifan Li
# Update: 20230529

from unicorn import *
import logging

# Page size required by Unicorn
UNICORN_PAGE_SIZE = 0x1000

# Max allowable segment size (1G)
MAX_ALLOWABLE_SEG_SIZE = 1024 * 1024 * 1024

# Alignment functions to align all memory segments to Unicorn page boundaries (4KB page
s only)
ALIGN_PAGE_DOWN = lambda x: x & ~(UNICORN_PAGE_SIZE - 1)
ALIGN_PAGE_UP   = lambda x: (x + UNICORN_PAGE_SIZE - 1) & ~(UNICORN_PAGE_SIZE 1)

# refer: https://github.com/Battelle/afl-unicorn/blob/master/unicorn_mode/helper_script
s/unicorn_loader.py
class UnicornSimpleHeap(object):
    """ Use this class to provide a simple heap implementation. This should
        be used if malloc/free calls break things during emulation. This heap also
        implements basic guard-page capabilities which enable immediate notice of
        heap overflow and underflows.
    """

    # Helper data-container used to track chunks
    class HeapChunk(object):
        def __init__(self, actual_addr, total_size, data_size):
            self.total_size = total_size                        # Total size of the chu
nk (including padding and guard page)
            self.actual_addr = actual_addr                      # Actual start address
of the chunk
            self.data_size = data_size                          # Size requested by the
 caller of actual malloc call
            self.data_addr = actual_addr + UNICORN_PAGE_SIZE    # Address where data ac
tually starts

        # Returns true if the specified buffer is completely within the chunk, else fal
se
        def is_buffer_in_chunk(self, addr, size):
            if addr >= self.data_addr and ((addr + size) <= (self.data_addr + self.data
```

```
_size)):
                return True
            else:
                return False

        def isSameChunk(self, anotherChunk):
            isSame = (self.actual_addr == anotherChunk.actual_addr) and (self.total_siz
e == anotherChunk.total_size)
            return isSame

        def debug(self):
            chunkEndAddr = self.actual_addr + self.total_size
            chunkStr = "chunk: [0x%X-0x%X] ptr=0x%X, size=%d=0x%X"% (self.actual_addr,
chunkEndAddr, self.data_addr, self.data_size, self.data_size)
            return chunkStr

        def isOverlapped(self, newChunk):
            # logging.info("debug: self=%s, newChunk=%s", self.debug(), newChunk.debug(
))
            selfStartAddr = self.actual_addr
            selfLastAddr = selfStartAddr + self.total_size - 1
            newChunkStartAddr = newChunk.actual_addr
            newChunkLastAddr = newChunkStartAddr + newChunk.total_size - 1
            isOverlapStart = (newChunkStartAddr >= selfStartAddr) and (newChunkStartAdd
r <= selfLastAddr)
            isOverlapEnd = (newChunkLastAddr >= selfStartAddr) and (newChunkLastAddr <=
 selfLastAddr)
            isOverlapped = isOverlapStart or isOverlapEnd
            return isOverlapped

    # # Skip the zero-page to avoid weird potential issues with segment registers
    # HEAP_MIN_ADDR = 0x00002000 # 8KB
    # HEAP_MAX_ADDR = 0xFFFFFFFF # 4GB-1
    _headMinAddr = None
    _heapMaxAddr = None

    _uc = None              # Unicorn engine instance to interact with
    _chunks = []            # List of all known chunks
    _debug_print = False    # True to print debug information

    # def __init__(self, uc, debug_print=False):
    def __init__(self, uc, headMinAddr, heapMaxAddr, debug_print=False):
        self._uc = uc
        self._headMinAddr = headMinAddr
        self._heapMaxAddr = heapMaxAddr
        self._debug_print = debug_print

        # Add the watchpoint hook that will be used to implement psuedo-guard page supp
ort
        self._uc.hook_add(UC_HOOK_MEM_WRITE | UC_HOOK_MEM_READ, self.__check_mem_access)


    def isChunkAllocated(self, newChunk):
        isAllocated = False
        for eachChunk in self._chunks:
            if eachChunk.isSameChunk(newChunk):
```

```
                isAllocated = True
                break
        return isAllocated

    def isChunkOverlapped(self, newChunk):
        isOverlapped = False
        for eachChunk in self._chunks:
            if eachChunk.isOverlapped(newChunk):
                isOverlapped = True
                break
        return isOverlapped

    def malloc(self, size):
        # Figure out the overall size to be allocated/mapped
        #    - Allocate at least 1 4k page of memory to make Unicorn happy
        #    - Add guard pages at the start and end of the region
        total_chunk_size = UNICORN_PAGE_SIZE + ALIGN_PAGE_UP(size) + UNICORN_PAGE_SIZE
        # Gross but efficient way to find space for the chunk:
        chunk = None
        # for addr in range(self.HEAP_MIN_ADDR, self.HEAP_MAX_ADDR, UNICORN_PAGE_SIZE):
        for addr in range(self._headMinAddr, self._heapMaxAddr, UNICORN_PAGE_SIZE):
            try:
                # self._uc.mem_map(addr, total_chunk_size, UC_PROT_READ | UC_PROT_WRITE)

                chunk = self.HeapChunk(addr, total_chunk_size, size)
                # chunkStr = "[0x{0:X}-0x{1:X}]".format(chunk.actual_addr, chunk.actual
_addr + chunk.total_size)
                chunkStr = chunk.debug()
                # if chunk in self._chunks:
                # if self.isChunkAllocated(chunk):
                if self.isChunkOverlapped(chunk):
                    # if self._debug_print:
                    #     logging.info(" ~~ Omit overlapped chunk: %s", chunkStr)
                    continue
                else:
                    if self._debug_print:
                        # logging.info("Heap: allocating 0x{0:X} byte addr=0x{1:X} of c
hunk {2:s}".format(chunk.data_size, chunk.data_addr, chunkStr))
                        logging.info(" ++ Allocated heap chunk: %s", chunkStr)
                    break
            except UcError as err:
                logging.error("!!! Heap malloc failed: error=%s", err)
                continue
        # Something went very wrong
        if chunk == None:
            return 0
        self._chunks.append(chunk)
        return chunk.data_addr

    def calloc(self, size, count):
        # Simple wrapper around malloc with calloc() args
        return self.malloc(size count)

    def realloc(self, ptr, new_size):
        # Wrapper around malloc(new_size) / memcpy(new, old, old_size) / free(old)
        if self._debug_print:
```

```
            logging.info("Reallocating chunk @ 0x{0:016x} to be 0x{1:x} bytes".format(p
tr, new_size))
        old_chunk = None
        for chunk in self._chunks:
            if chunk.data_addr == ptr:
                old_chunk = chunk
        new_chunk_addr = self.malloc(new_size)
        if old_chunk != None:
            self._uc.mem_write(new_chunk_addr, str(self._uc.mem_read(old_chunk.data_addr
, old_chunk.data_size)))
            self.free(old_chunk.data_addr)
        return new_chunk_addr

    def free(self, addr):
        for chunk in self._chunks:
            if chunk.is_buffer_in_chunk(addr, 1):
                if self._debug_print:
                    logging.info("Freeing 0x{0:x}-byte chunk @ 0x{0:016x}".format(chunk.
req_size, chunk.data_addr))
                self._uc.mem_unmap(chunk.actual_addr, chunk.total_size)
                self._chunks.remove(chunk)
                return True
        return False

    # Implements basic guard-page functionality
    def __check_mem_access(self, uc, access, address, size, value, user_data):
        for chunk in self._chunks:
            if address >= chunk.actual_addr and ((address + size) <= (chunk.actual_addr
+ chunk.total_size)):
                if chunk.is_buffer_in_chunk(address, size) == False:
                    if self._debug_print:
                        logging.info("Heap over/underflow attempting to {0} 0x{1:x} byt
es @ {2:016x}".format( \
                            "write" if access == UC_MEM_WRITE else "read", size, address
))
                    # Force a memory-based crash
                    uc.force_crash(UcError(UC_ERR_READ_PROT))
```

辅助代码：`crifanLogging.py`

如果用Python代码（去使用Unicorn），则其中的日志打印，可以参考：`crifanLibPython.py`

其具体完整的最新的代码，可从crifanLibPython中下载到源码：

- crifanLogging.py

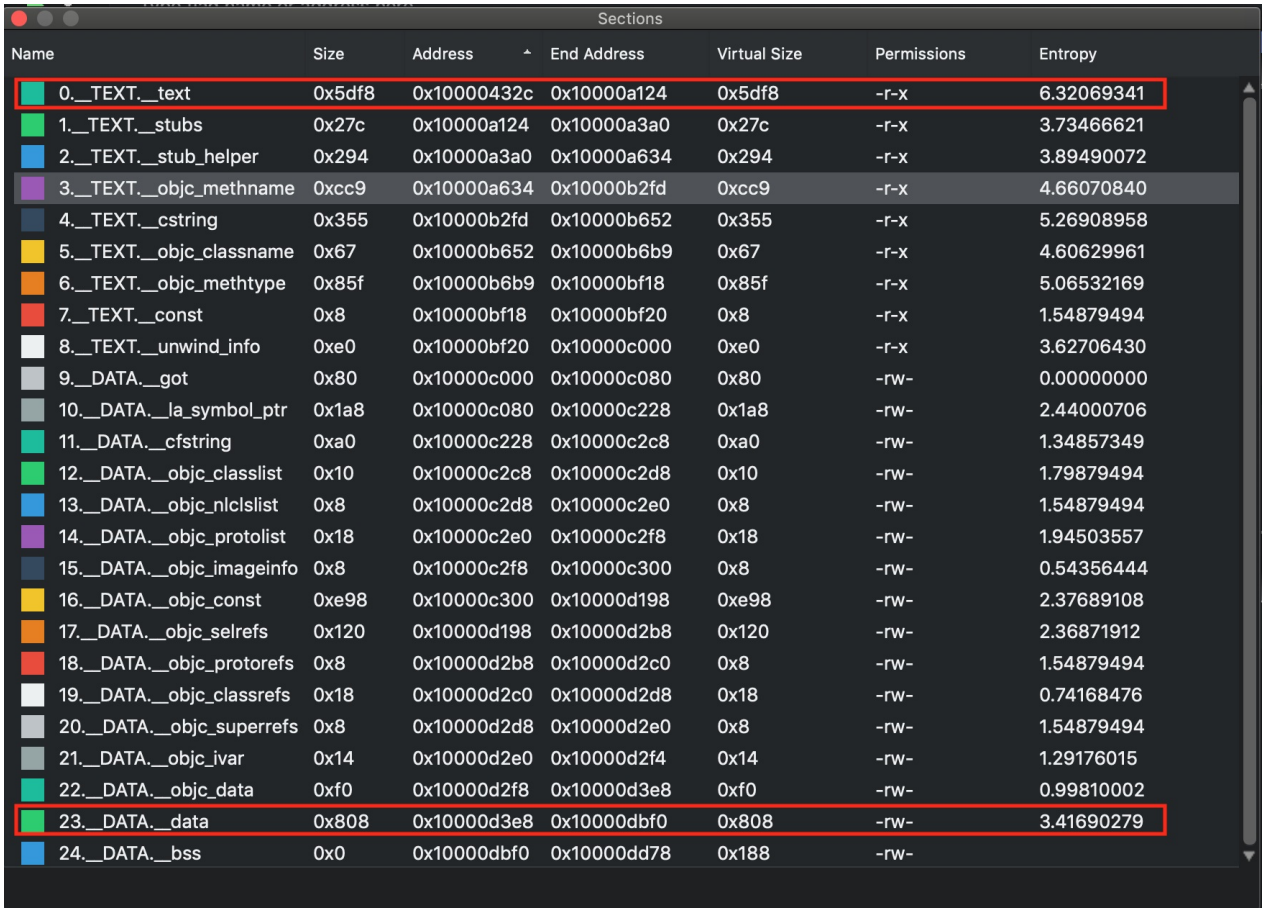
其他示例代码

iOS Tampering and Reverse Engineering

- iOS Tampering and Reverse Engineering - OWASP Mobile Application Security

```
import lief
from unicorn import *
from unicorn.arm64_const import *

# --- Extract __text and __data section content from the binary ---
binary = lief.parse("uncrackable.arm64")
text_section = binary.get_section("__text")
text_content = text_section.content

data_section = binary.get_section("__data")
data_content = data_section.content

# --- Setup Unicorn for ARM64 execution ---
arch = "arm64le"
emu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)

# --- Create Stack memory ---
addr = 0x40000000
size = 1024 * 1024
emu.mem_map(addr, size)
emu.reg_write(UC_ARM64_REG_SP, addr + size - 1)

# --- Load text section ---
base_addr = 0x100000000
tmp_len = 1024 * 1024
text_section_load_addr = 0x10000432c
emu.mem_map(base_addr, tmp_len)
emu.mem_write(text_section_load_addr, bytes(text_content))

# --- Load data section ---
data_section_load_addr = 0x10000d3e8
emu.mem_write(data_section_load_addr, bytes(data_content))

# --- Hack for stack_chk_guard ---
# without this will throw invalid memory read at 0x0
emu.mem_map(0x0, 1024)
emu.mem_write(0x0, b"00")

# --- Execute from 0x1000080d4 to 0x100008154 ---
emu.emu_start(0x1000080d4, 0x100008154)
ret_value = emu.reg_read(UC_ARM64_REG_X0)

# --- Dump return value ---
print(emu.mem_read(ret_value, 11))
```

其中，text和data的地址，来自于Mach-O格式分析的结果：

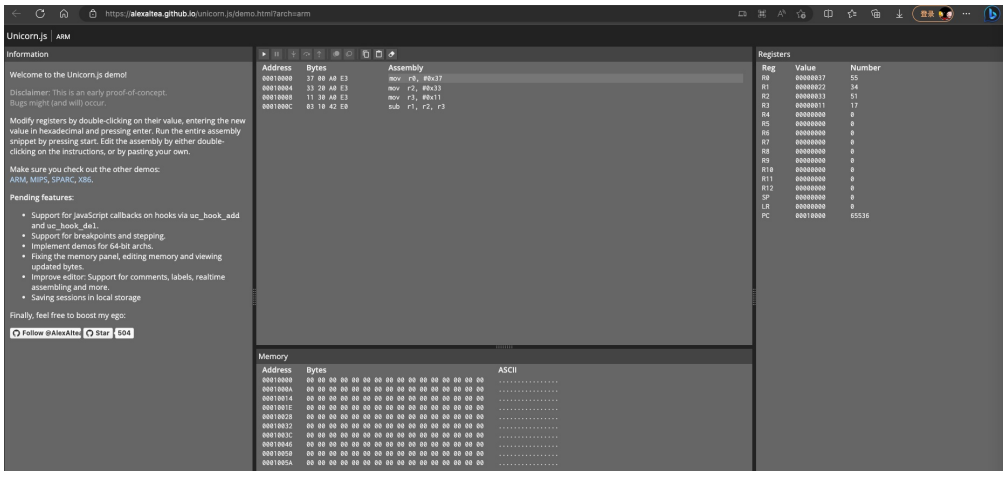


附录

下面列出相关参考资料。

Unicorn文档和资料

- Unicorn文档和资料
  - 主页
    - GitHub
      - Unicorn CPU emulator framework (ARM, AArch64, M68K, Mips, Sparc, X86)
        - https://github.com/unicorn-engine/unicorn
    - 官网
      - Unicorn – The ultimate CPU emulator
        - http://www.unicorn-engine.org
  - 官网文档
    - Documentation – Unicorn – The Ultimate CPU emulator (unicorn-engine.org)
    - 入门教程：C和Python
      - Programming with C & Python languages – Unicorn – The Ultimate CPU emulator (unicorn-engine.org)
  - 其他实例代码
    - C
      - unicorn/samples at master · unicorn-engine/unicorn · GitHub
        - arm
          - https://github.com/unicorn-engine/unicorn/blob/master/samples/sample_arm.c
          - https://github.com/unicorn-engine/unicorn/blob/master/samples/sample_arm64.c
        - x86
          - https://github.com/unicorn-engine/unicorn/blob/master/samples/sample_x86.c
    - Python
      - unicorn/bindings/python at master · unicorn-engine/unicorn · GitHub
        - arm
          - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_arm.py
          - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_arm64.py
          - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_arm64eb.py
          - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_armeb.py
        - x86
          - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_x86.py
  - （Unicorn Engine的）API接口文档
    - 非官方
      - C
        - 中文
          - Unicorn-Engine-Documentation/Unicorn-Engine Documentation.md at master · kabeor/Unicorn-Engine-Documentation (github.com)

        - 英文
            - Unicorn Engine Reference (Unofficial) - HackMD
    - FAQ
        - unicorn/FAQ.md at master · unicorn-engine/unicorn · GitHub
    - 其他
        - Unicorn和QEMU的关系和对比
            - Unicorn & QEMU – Unicorn – The Ultimate CPU emulator (unicorn-engine.org)
        - Unicorn实现细节
            - PDF: 《BlackHat USA 2015 slides》 == Unicorn: next generation CPU emulator framework
                - https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf
                - https://github.com/unicorn-engine/unicorn/blob/master/docs/BHUSA2015-unicorn.pdf
        - 自己编译
            - unicorn/COMPILE.md at master · unicorn-engine/unicorn · GitHub
        - 部分网上资源
            - Running arm64 code on your Intel Mac using Unicorn emulator | danylokos's blog
            - Unicorn的一些笔记和用法
                - alexander-hanel/unicorn-engine-notes: Notes on using the Python bindings for the Unicorn Engine (github.com)
            - hook函数写法
                - debug_utils.py
                    - AndroidNativeEmu/debug_utils.py at 8568b316720c6cb543be32562491b3fc172dc6c0 · AeonLucid/AndroidNativeEmu · GitHub
                - main.py
                    - frick/main.py at 5a60bc8fe8ddcac146853ad22b64e3bac7930269 · iGio90/frick (github.com)
    - 在线工具
        - 在线模拟unicorn去运行arm汇编代码
            - Unicorn.js: ARM (alexaltea.github.io)
            - 

Unicorn部分核心代码

unicorn.h

- unicorn.h
  - unicorn/unicorn.h at master · unicorn-engine/unicorn · GitHub
    - https://github.com/unicorn-engine/unicorn/blob/master/include/unicorn/unicorn.h

arch架构

```
// Architecture type
typedef enum uc_arch {
    UC_ARCH_ARM = 1, // ARM architecture (including Thumb, Thumb-2)
    UC_ARCH_ARM64,   // ARM-64, also called AArch64
    UC_ARCH_MIPS,    // Mips architecture
    UC_ARCH_X86,     // X86 architecture (including x86 & x86-64)
    UC_ARCH_PPC,     // PowerPC architecture
    UC_ARCH_SPARC,   // Sparc architecture
    UC_ARCH_M68K,    // M68K architecture
    UC_ARCH_RISCV,   // RISCV architecture
    UC_ARCH_S390X,   // S390X architecture
    UC_ARCH_TRICORE, // TriCore architecture
    UC_ARCH_MAX,
} uc_arch;
```

mode模式

```
// Mode type
typedef enum uc_mode {
    UC_MODE_LITTLE_ENDIAN = 0,    // little-endian mode (default mode)
    UC_MODE_BIG_ENDIAN = 1 << 30, // big-endian mode

    // arm / arm64
    UC_MODE_ARM = 0,        // ARM mode
    UC_MODE_THUMB = 1 << 4, // THUMB mode (including Thumb-2)
    // Depreciated, use UC_ARM_CPU_* with uc_ctl instead.
    UC_MODE_MCLASS = 1 << 5,  // ARM's Cortex-M series.
    UC_MODE_V8 = 1 << 6,      // ARMv8 A32 encodings for ARM
    UC_MODE_ARMBE8 = 1 << 10, // Big-endian data and Little-endian code.
                              // Legacy support for UC1 only.

    // arm (32bit) cpu types
    // Depreciated, use UC_ARM_CPU_* with uc_ctl instead.
    UC_MODE_ARM926 = 1 << 7,  // ARM926 CPU type
    UC_MODE_ARM946 = 1 << 8,  // ARM946 CPU type
    UC_MODE_ARM1176 = 1 << 9, // ARM1176 CPU type

    // mips
    UC_MODE_MICRO = 1 << 4,    // MicroMips mode (currently unsupported)
```

```
    UC_MODE_MIPS3 = 1 << 5,     // Mips III ISA (currently unsupported)
    UC_MODE_MIPS32R6 = 1 << 6, // Mips32r6 ISA (currently unsupported)
    UC_MODE_MIPS32 = 1 << 2,    // Mips32 ISA
    UC_MODE_MIPS64 = 1 << 3,    // Mips64 ISA

    // x86 / x64
    UC_MODE_16 = 1 << 1, // 16-bit mode
    UC_MODE_32 = 1 << 2, // 32-bit mode
    UC_MODE_64 = 1 << 3, // 64-bit mode

    // ppc
    UC_MODE_PPC32 = 1 << 2, // 32-bit mode
    UC_MODE_PPC64 = 1 << 3, // 64-bit mode (currently unsupported)
    UC_MODE_QPX =
        1 << 4, // Quad Processing eXtensions mode (currently unsupported)

    // sparc
    UC_MODE_SPARC32 = 1 << 2, // 32-bit mode
    UC_MODE_SPARC64 = 1 << 3, // 64-bit mode
    UC_MODE_V9 = 1 << 4,      // SparcV9 mode (currently unsupported)

    // riscv
    UC_MODE_RISCV32 = 1 << 2, // 32-bit mode
    UC_MODE_RISCV64 = 1 << 3, // 64-bit mode

    // m68k
} uc_mode;
```

错误类型

```
// All type of errors encountered by Unicorn API.
// These are values returned by uc_errno()
typedef enum uc_err {
    UC_ERR_OK = 0,          // No error: everything was fine
    UC_ERR_NOMEM,           // Out-Of-Memory error: uc_open(), uc_emulate()
    UC_ERR_ARCH,            // Unsupported architecture: uc_open()
    UC_ERR_HANDLE,          // Invalid handle
    UC_ERR_MODE,            // Invalid/unsupported mode: uc_open()
    UC_ERR_VERSION,         // Unsupported version (bindings)
    UC_ERR_READ_UNMAPPED,   // Quit emulation due to READ on unmapped memory:
                            // uc_emu_start()
    UC_ERR_WRITE_UNMAPPED,  // Quit emulation due to WRITE on unmapped memory:
                            // uc_emu_start()
    UC_ERR_FETCH_UNMAPPED,  // Quit emulation due to FETCH on unmapped memory:
                            // uc_emu_start()
    UC_ERR_HOOK,            // Invalid hook type: uc_hook_add()
    UC_ERR_INSN_INVALID,    // Quit emulation due to invalid instruction:
                            // uc_emu_start()
    UC_ERR_MAP,             // Invalid memory mapping: uc_mem_map()
    UC_ERR_WRITE_PROT,      // Quit emulation due to UC_MEM_WRITE_PROT violation:
                            // uc_emu_start()
    UC_ERR_READ_PROT,       // Quit emulation due to UC_MEM_READ_PROT violation:
                            // uc_emu_start()
    UC_ERR_FETCH_PROT,      // Quit emulation due to UC_MEM_FETCH_PROT violation:
```

```
                            // uc_emu_start()
    UC_ERR_ARG, // Inavalid argument provided to uc_xxx function (See specific
                // function API)
    UC_ERR_READ_UNALIGNED,   // Unaligned read
    UC_ERR_WRITE_UNALIGNED, // Unaligned write
    UC_ERR_FETCH_UNALIGNED, // Unaligned fetch
    UC_ERR_HOOK_EXIST,       // hook for this event already existed
    UC_ERR_RESOURCE,         // Insufficient resource: uc_emu_start()
    UC_ERR_EXCEPTION,        // Unhandled CPU exception
} uc_err;
```

hook类型

```
// All type of hooks for uc_hook_add() API.
typedef enum uc_hook_type {
    // Hook all interrupt/syscall events
    UC_HOOK_INTR = 1 << 0,
    // Hook a particular instruction - only a very small subset of instructions
    // supported here
    UC_HOOK_INSN = 1 << 1,
    // Hook a range of code
    UC_HOOK_CODE = 1 << 2,
    // Hook basic blocks
    UC_HOOK_BLOCK = 1 << 3,
    // Hook for memory read on unmapped memory
    UC_HOOK_MEM_READ_UNMAPPED = 1 << 4,
    // Hook for invalid memory write events
    UC_HOOK_MEM_WRITE_UNMAPPED = 1 << 5,
    // Hook for invalid memory fetch for execution events
    UC_HOOK_MEM_FETCH_UNMAPPED = 1 << 6,
    // Hook for memory read on read-protected memory
    UC_HOOK_MEM_READ_PROT = 1 << 7,
    // Hook for memory write on write-protected memory
    UC_HOOK_MEM_WRITE_PROT = 1 << 8,
    // Hook for memory fetch on non-executable memory
    UC_HOOK_MEM_FETCH_PROT = 1 << 9,
    // Hook memory read events.
    UC_HOOK_MEM_READ = 1 << 10,
    // Hook memory write events.
    UC_HOOK_MEM_WRITE = 1 << 11,
    // Hook memory fetch for execution events
    UC_HOOK_MEM_FETCH = 1 << 12,
    // Hook memory read events, but only successful access.
    // The callback will be triggered after successful read.
    UC_HOOK_MEM_READ_AFTER = 1 << 13,
    // Hook invalid instructions exceptions.
    UC_HOOK_INSN_INVALID = 1 << 14,
    // Hook on new edge generation. Could be useful in program analysis.
    //
    // NOTE: This is different from UC_HOOK_BLOCK in 2 ways:
    //       1. The hook is called before executing code.
    //       2. The hook is only called when generation is triggered.
    UC_HOOK_EDGE_GENERATED = 1 << 15,
    // Hook on specific tcg op code. The usage of this hook is similar to
```

```
    // UC_HOOK_INSN.
    UC_HOOK_TCG_OPCODE = 1 << 16,
} uc_hook_type;
```

unicorn.py

- unicorn.py
  - unicorn/unicorn.py at master · unicorn-engine/unicorn · GitHub
    - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/unicorn/unicorn.py

主要函数

```
_setup_prototype(_uc, "uc_version", ctypes.c_uint, ctypes.POINTER(ctypes.c_int), ctypes.
POINTER(ctypes.c_int))
_setup_prototype(_uc, "uc_arch_supported", ctypes.c_bool, ctypes.c_int)
_setup_prototype(_uc, "uc_open", ucerr, ctypes.c_uint, ctypes.c_uint, ctypes.POINTER(uc
_engine))
_setup_prototype(_uc, "uc_close", ucerr, uc_engine)
_setup_prototype(_uc, "uc_strerror", ctypes.c_char_p, ucerr)
_setup_prototype(_uc, "uc_errno", ucerr, uc_engine)
_setup_prototype(_uc, "uc_reg_read", ucerr, uc_engine, ctypes.c_int, ctypes.c_void_p)
_setup_prototype(_uc, "uc_reg_write", ucerr, uc_engine, ctypes.c_int, ctypes.c_void_p)
_setup_prototype(_uc, "uc_mem_read", ucerr, uc_engine, ctypes.c_uint64, ctypes.POINTER(
ctypes.c_char), ctypes.c_size_t)
_setup_prototype(_uc, "uc_mem_write", ucerr, uc_engine, ctypes.c_uint64, ctypes.POINTER(
ctypes.c_char), ctypes.c_size_t)
_setup_prototype(_uc, "uc_emu_start", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_uint64
, ctypes.c_uint64, ctypes.c_size_t)
_setup_prototype(_uc, "uc_emu_stop", ucerr, uc_engine)
_setup_prototype(_uc, "uc_hook_del", ucerr, uc_engine, uc_hook_h)
_setup_prototype(_uc, "uc_mmio_map", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_size_t,
 ctypes.c_void_p, ctypes.c_void_p, ctypes.c_void_p, ctypes.c_void_p)
_setup_prototype(_uc, "uc_mem_map", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_size_t,
 ctypes.c_uint32)
_setup_prototype(_uc, "uc_mem_map_ptr", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_siz
e_t, ctypes.c_uint32, ctypes.c_void_p)
_setup_prototype(_uc, "uc_mem_unmap", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_size_t
)
_setup_prototype(_uc, "uc_mem_protect", ucerr, uc_engine, ctypes.c_uint64, ctypes.c_siz
e_t, ctypes.c_uint32)
_setup_prototype(_uc, "uc_query", ucerr, uc_engine, ctypes.c_uint32, ctypes.POINTER(cty
pes.c_size_t))
_setup_prototype(_uc, "uc_context_alloc", ucerr, uc_engine, ctypes.POINTER(uc_context))
_setup_prototype(_uc, "uc_free", ucerr, ctypes.c_void_p)
_setup_prototype(_uc, "uc_context_save", ucerr, uc_engine, uc_context)
_setup_prototype(_uc, "uc_context_restore", ucerr, uc_engine, uc_context)
_setup_prototype(_uc, "uc_context_size", ctypes.c_size_t, uc_engine)
_setup_prototype(_uc, "uc_context_reg_read", ucerr, uc_context, ctypes.c_int, ctypes.c_
void_p)
_setup_prototype(_uc, "uc_context_reg_write", ucerr, uc_context, ctypes.c_int, ctypes.c
_void_p)
_setup_prototype(_uc, "uc_context_free", ucerr, uc_context)
_setup_prototype(_uc, "uc_mem_regions", ucerr, uc_engine, ctypes.POINTER(ctypes.POINTER(
```

```
_uc_mem_region)), ctypes.POINTER(ctypes.c_uint32))
# https://bugs.python.org/issue42880
_setup_prototype(_uc, "uc_hook_add", ucerr, uc_engine, ctypes.POINTER(uc_hook_h), ctypes
.c_int, ctypes.c_void_p, ctypes.c_void_p, ctypes.c_uint64, ctypes.c_uint64)
_setup_prototype(_uc, "uc_ctl", ucerr, uc_engine, ctypes.c_int)
```

uc_priv.h

- uc_priv.h
  - unicorn/uc_priv.h at master · unicorn-engine/unicorn · GitHub
    - https://github.com/unicorn-engine/unicorn/blob/master/include/uc_priv.h

hook的结构体

```
struct hook {
    int type;       // UC_HOOK_*
    int insn;       // instruction for HOOK_INSN
    int refs;       // reference count to free hook stored in multiple lists
    int op;         // opcode for HOOK_TCG_OPCODE
    int op_flags;   // opcode flags for HOOK_TCG_OPCODE
    bool to_delete; // set to true when the hook is deleted by the user. The
                    // destruction of the hook is delayed.
    uint64_t begin, end; // only trigger if PC or memory access is in this
                         // address (depends on hook type)
    void *callback;      // a uc_cb_* type
    void *user_data;
    GHashTable *hooked_regions; // The regions this hook instrumented on
};
```

hook列表

```
// hook list offsets
//
// The lowest 6 bits are used for hook type index while the others
// are used for hook flags.
//
// mirrors the order of uc_hook_type from include/unicorn/unicorn.h
typedef enum uc_hook_idx {
    UC_HOOK_INTR_IDX,
    UC_HOOK_INSN_IDX,
    UC_HOOK_CODE_IDX,
    UC_HOOK_BLOCK_IDX,
    UC_HOOK_MEM_READ_UNMAPPED_IDX,
    UC_HOOK_MEM_WRITE_UNMAPPED_IDX,
    UC_HOOK_MEM_FETCH_UNMAPPED_IDX,
    UC_HOOK_MEM_READ_PROT_IDX,
    UC_HOOK_MEM_WRITE_PROT_IDX,
    UC_HOOK_MEM_FETCH_PROT_IDX,
    UC_HOOK_MEM_READ_IDX,
    UC_HOOK_MEM_WRITE_IDX,
```

```
    UC_HOOK_MEM_FETCH_IDX,
    UC_HOOK_MEM_READ_AFTER_IDX,
    UC_HOOK_INSN_INVALID_IDX,
    UC_HOOK_EDGE_GENERATED_IDX,
    UC_HOOK_TCG_OPCODE_IDX,

    UC_HOOK_MAX,
} uc_hook_idx;
```

unicorn对象的结构体

```
struct uc_struct {
    uc_arch arch;
    uc_mode mode;
    uc_err errnum; // qemu/cpu-exec.c
    AddressSpace address_space_memory;
    AddressSpace address_space_io;
    query_t query;
    reg_read_t reg_read;
    reg_write_t reg_write;
    reg_reset_t reg_reset;

    uc_write_mem_t write_mem;
    uc_read_mem_t read_mem;
    uc_args_void_t release;  // release resource when uc_close()
    uc_args_uc_u64_t set_pc; // set PC for tracecode
    uc_get_pc_t get_pc;
    uc_args_int_t
        stop_interrupt; // check if the interrupt should stop emulation
    uc_memory_map_io_t memory_map_io;

    uc_args_uc_t init_arch, cpu_exec_init_all;
    uc_args_int_uc_t vm_start;
    uc_args_uc_long_t tcg_exec_init;
    uc_args_uc_ram_size_t memory_map;
    uc_args_uc_ram_size_ptr_t memory_map_ptr;
    uc_mem_unmap_t memory_unmap;
    uc_readonly_mem_t readonly_mem;
    uc_mem_redirect_t mem_redirect;
    uc_cpus_init cpus_init;
    uc_target_page_init target_page;
    uc_softfloat_initialize softfloat_initialize;
    uc_tcg_flush_tlb tcg_flush_tlb;
    uc_invalidate_tb_t uc_invalidate_tb;
    uc_gen_tb_t uc_gen_tb;
    uc_tb_flush_t tb_flush;
    uc_add_inline_hook_t add_inline_hook;
    uc_del_inline_hook_t del_inline_hook;

    uc_context_size_t context_size;
    uc_context_save_t context_save;
    uc_context_restore_t context_restore;

    /*  only 1 cpu in unicorn,
```

```
    do not need current_cpu to handle current running cpu. */
CPUState *cpu;

uc_insn_hook_validate insn_hook_validate;
uc_opcode_hook_validate_t opcode_hook_invalidate;

MemoryRegion *system_memory;     // qemu/exec.c
MemoryRegion *system_io;         // qemu/exec.c
MemoryRegion io_mem_unassigned; // qemu/exec.c
RAMList ram_list;                // qemu/exec.c
/* qemu/exec.c */
unsigned int alloc_hint;
/* qemu/exec-vary.c */
TargetPageBits *init_target_page;
int target_bits; // User defined page bits by uc_ctl
int cpu_model;
BounceBuffer bounce;                 // qemu/cpu-exec.c
volatile sig_atomic_t exit_request; // qemu/cpu-exec.c
/* qemu/accel/tcg/cpu-exec-common.c */
/* always be true after call tcg_exec_init(). */
bool tcg_allowed;
/* This is a multi-level map on the virtual address space.
   The bottom level has pointers to PageDesc.  */
void **l1_map; // qemu/accel/tcg/translate-all.c
size_t l1_map_size;
/* qemu/accel/tcg/translate-all.c */
int v_l1_size;
int v_l1_shift;
int v_l2_levels;
/* code generation context */
TCGContext *tcg_ctx;
/* memory.c */
QTAILQ_HEAD(memory_listeners, MemoryListener) memory_listeners;
QTAILQ_HEAD(, AddressSpace) address_spaces;
GHashTable *flat_views;
bool memory_region_update_pending;

// linked lists containing hooks per type
struct list hook[UC_HOOK_MAX];
struct list hooks_to_del;
int hooks_count[UC_HOOK_MAX];

// hook to count number of instructions for uc_emu_start()
uc_hook count_hook;

size_t emu_counter; // current counter of uc_emu_start()
size_t emu_count;   // save counter of uc_emu_start()

int size_recur_mem; // size for mem access when in a recursive call

bool init_tcg;       // already initialized local TCGv variables?
bool stop_request;   // request to immediately stop emulation - for
                     // uc_emu_stop()
bool quit_request;   // request to quit the current TB, but continue to
                     // emulate - for uc_mem_protect()
bool emulation_done; // emulation is done by uc_emu_start()
```

```
    bool timed_out;      // emulation timed out, that can retrieve via
                         // uc_query(UC_QUERY_TIMEOUT)
    QemuThread timer;    // timer for emulation timeout
    uint64_t timeout;    // timeout for uc_emu_start()

    uint64_t invalid_addr; // invalid address to be accessed
    int invalid_error;     // invalid memory code: 1 = READ, 2 = WRITE, 3 = CODE

    int use_exits;
    uint64_t exits[UC_MAX_NESTED_LEVEL]; // When multiple exits is not enabled.
    GTree *ctl_exits; // addresses where emulation stops (@until param of
                      // uc_emu_start()) Also see UC_CTL_USE_EXITS for more
                      // details.

    int thumb; // thumb mode for ARM
    MemoryRegion **mapped_blocks;
    uint32_t mapped_block_count;
    uint32_t mapped_block_cache_index;
    void *qemu_thread_data; // to support cross compile to Windows
                            // (qemu-thread-win32.c)
    uint32_t target_page_size;
    uint32_t target_page_align;
    uint64_t qemu_host_page_size;
    uint64_t qemu_real_host_page_size;
    int qemu_icache_linesize;
    /* ARCH_REGS_STORAGE_SIZE */
    int cpu_context_size;
    uint64_t next_pc; // save next PC for some special cases
    bool hook_insert; // insert new hook at begin of the hook list (append by
                      // default)
    bool first_tb; // is this the first Translation-Block ever generated since
                   // uc_emu_start()?
    bool no_exit_request; // Disable check_exit_request temporarily. A
                          // workaround to treat the IT block as a whole block.
    bool init_done;       // Whether the initialization is done.

    sigjmp_buf jmp_bufs[UC_MAX_NESTED_LEVEL]; // To support nested uc_emu_start
    int nested_level;                         // Current nested_level

    struct TranslationBlock *last_tb; // The real last tb we executed.

    FlatView *empty_view; // Static function variable moved from flatviews_init
};
```

arm64.h

- arm64.h
  - unicorn/arm64.h at master · unicorn-engine/unicorn · GitHub
    - https://github.com/unicorn-engine/unicorn/blob/master/include/unicorn/arm64.h

arm64寄存器

```
//> ARM64 registers
```

```
typedef enum uc_arm64_reg {
    UC_ARM64_REG_INVALID = 0,

    UC_ARM64_REG_X29,
    UC_ARM64_REG_X30,
    UC_ARM64_REG_NZCV,
    UC_ARM64_REG_SP,
    UC_ARM64_REG_WSP,
    UC_ARM64_REG_WZR,
    UC_ARM64_REG_XZR,
    UC_ARM64_REG_B0,
    UC_ARM64_REG_B1,
    UC_ARM64_REG_B2,
    UC_ARM64_REG_B3,
    UC_ARM64_REG_B4,
    UC_ARM64_REG_B5,
    UC_ARM64_REG_B6,
    UC_ARM64_REG_B7,
    UC_ARM64_REG_B8,
    UC_ARM64_REG_B9,
    UC_ARM64_REG_B10,
    UC_ARM64_REG_B11,
    UC_ARM64_REG_B12,
    UC_ARM64_REG_B13,
    UC_ARM64_REG_B14,
    UC_ARM64_REG_B15,
    UC_ARM64_REG_B16,
    UC_ARM64_REG_B17,
    UC_ARM64_REG_B18,
    UC_ARM64_REG_B19,
    UC_ARM64_REG_B20,
    UC_ARM64_REG_B21,
    UC_ARM64_REG_B22,
    UC_ARM64_REG_B23,
    UC_ARM64_REG_B24,
    UC_ARM64_REG_B25,
    UC_ARM64_REG_B26,
    UC_ARM64_REG_B27,
    UC_ARM64_REG_B28,
    UC_ARM64_REG_B29,
    UC_ARM64_REG_B30,
    UC_ARM64_REG_B31,
    UC_ARM64_REG_D0,
    UC_ARM64_REG_D1,
    UC_ARM64_REG_D2,
    UC_ARM64_REG_D3,
    UC_ARM64_REG_D4,
    UC_ARM64_REG_D5,
    UC_ARM64_REG_D6,
    UC_ARM64_REG_D7,
    UC_ARM64_REG_D8,
    UC_ARM64_REG_D9,
    UC_ARM64_REG_D10,
    UC_ARM64_REG_D11,
    UC_ARM64_REG_D12,
    UC_ARM64_REG_D13,
```

```
    UC_ARM64_REG_D14,
    UC_ARM64_REG_D15,
    UC_ARM64_REG_D16,
    UC_ARM64_REG_D17,
    UC_ARM64_REG_D18,
    UC_ARM64_REG_D19,
    UC_ARM64_REG_D20,
    UC_ARM64_REG_D21,
    UC_ARM64_REG_D22,
    UC_ARM64_REG_D23,
    UC_ARM64_REG_D24,
    UC_ARM64_REG_D25,
    UC_ARM64_REG_D26,
    UC_ARM64_REG_D27,
    UC_ARM64_REG_D28,
    UC_ARM64_REG_D29,
    UC_ARM64_REG_D30,
    UC_ARM64_REG_D31,
    UC_ARM64_REG_H0,
    UC_ARM64_REG_H1,
    UC_ARM64_REG_H2,
    UC_ARM64_REG_H3,
    UC_ARM64_REG_H4,
    UC_ARM64_REG_H5,
    UC_ARM64_REG_H6,
    UC_ARM64_REG_H7,
    UC_ARM64_REG_H8,
    UC_ARM64_REG_H9,
    UC_ARM64_REG_H10,
    UC_ARM64_REG_H11,
    UC_ARM64_REG_H12,
    UC_ARM64_REG_H13,
    UC_ARM64_REG_H14,
    UC_ARM64_REG_H15,
    UC_ARM64_REG_H16,
    UC_ARM64_REG_H17,
    UC_ARM64_REG_H18,
    UC_ARM64_REG_H19,
    UC_ARM64_REG_H20,
    UC_ARM64_REG_H21,
    UC_ARM64_REG_H22,
    UC_ARM64_REG_H23,
    UC_ARM64_REG_H24,
    UC_ARM64_REG_H25,
    UC_ARM64_REG_H26,
    UC_ARM64_REG_H27,
    UC_ARM64_REG_H28,
    UC_ARM64_REG_H29,
    UC_ARM64_REG_H30,
    UC_ARM64_REG_H31,
    UC_ARM64_REG_Q0,
    UC_ARM64_REG_Q1,
    UC_ARM64_REG_Q2,
    UC_ARM64_REG_Q3,
    UC_ARM64_REG_Q4,
    UC_ARM64_REG_Q5,
```

```
    UC_ARM64_REG_Q6,
    UC_ARM64_REG_Q7,
    UC_ARM64_REG_Q8,
    UC_ARM64_REG_Q9,
    UC_ARM64_REG_Q10,
    UC_ARM64_REG_Q11,
    UC_ARM64_REG_Q12,
    UC_ARM64_REG_Q13,
    UC_ARM64_REG_Q14,
    UC_ARM64_REG_Q15,
    UC_ARM64_REG_Q16,
    UC_ARM64_REG_Q17,
    UC_ARM64_REG_Q18,
    UC_ARM64_REG_Q19,
    UC_ARM64_REG_Q20,
    UC_ARM64_REG_Q21,
    UC_ARM64_REG_Q22,
    UC_ARM64_REG_Q23,
    UC_ARM64_REG_Q24,
    UC_ARM64_REG_Q25,
    UC_ARM64_REG_Q26,
    UC_ARM64_REG_Q27,
    UC_ARM64_REG_Q28,
    UC_ARM64_REG_Q29,
    UC_ARM64_REG_Q30,
    UC_ARM64_REG_Q31,
    UC_ARM64_REG_S0,
    UC_ARM64_REG_S1,
    UC_ARM64_REG_S2,
    UC_ARM64_REG_S3,
    UC_ARM64_REG_S4,
    UC_ARM64_REG_S5,
    UC_ARM64_REG_S6,
    UC_ARM64_REG_S7,
    UC_ARM64_REG_S8,
    UC_ARM64_REG_S9,
    UC_ARM64_REG_S10,
    UC_ARM64_REG_S11,
    UC_ARM64_REG_S12,
    UC_ARM64_REG_S13,
    UC_ARM64_REG_S14,
    UC_ARM64_REG_S15,
    UC_ARM64_REG_S16,
    UC_ARM64_REG_S17,
    UC_ARM64_REG_S18,
    UC_ARM64_REG_S19,
    UC_ARM64_REG_S20,
    UC_ARM64_REG_S21,
    UC_ARM64_REG_S22,
    UC_ARM64_REG_S23,
    UC_ARM64_REG_S24,
    UC_ARM64_REG_S25,
    UC_ARM64_REG_S26,
    UC_ARM64_REG_S27,
    UC_ARM64_REG_S28,
    UC_ARM64_REG_S29,
```

```
    UC_ARM64_REG_S30,
    UC_ARM64_REG_S31,
    UC_ARM64_REG_W0,
    UC_ARM64_REG_W1,
    UC_ARM64_REG_W2,
    UC_ARM64_REG_W3,
    UC_ARM64_REG_W4,
    UC_ARM64_REG_W5,
    UC_ARM64_REG_W6,
    UC_ARM64_REG_W7,
    UC_ARM64_REG_W8,
    UC_ARM64_REG_W9,
    UC_ARM64_REG_W10,
    UC_ARM64_REG_W11,
    UC_ARM64_REG_W12,
    UC_ARM64_REG_W13,
    UC_ARM64_REG_W14,
    UC_ARM64_REG_W15,
    UC_ARM64_REG_W16,
    UC_ARM64_REG_W17,
    UC_ARM64_REG_W18,
    UC_ARM64_REG_W19,
    UC_ARM64_REG_W20,
    UC_ARM64_REG_W21,
    UC_ARM64_REG_W22,
    UC_ARM64_REG_W23,
    UC_ARM64_REG_W24,
    UC_ARM64_REG_W25,
    UC_ARM64_REG_W26,
    UC_ARM64_REG_W27,
    UC_ARM64_REG_W28,
    UC_ARM64_REG_W29,
    UC_ARM64_REG_W30,
    UC_ARM64_REG_X0,
    UC_ARM64_REG_X1,
    UC_ARM64_REG_X2,
    UC_ARM64_REG_X3,
    UC_ARM64_REG_X4,
    UC_ARM64_REG_X5,
    UC_ARM64_REG_X6,
    UC_ARM64_REG_X7,
    UC_ARM64_REG_X8,
    UC_ARM64_REG_X9,
    UC_ARM64_REG_X10,
    UC_ARM64_REG_X11,
    UC_ARM64_REG_X12,
    UC_ARM64_REG_X13,
    UC_ARM64_REG_X14,
    UC_ARM64_REG_X15,
    UC_ARM64_REG_X16,
    UC_ARM64_REG_X17,
    UC_ARM64_REG_X18,
    UC_ARM64_REG_X19,
    UC_ARM64_REG_X20,
    UC_ARM64_REG_X21,
    UC_ARM64_REG_X22,
```

```
    UC_ARM64_REG_X23,
    UC_ARM64_REG_X24,
    UC_ARM64_REG_X25,
    UC_ARM64_REG_X26,
    UC_ARM64_REG_X27,
    UC_ARM64_REG_X28,

    UC_ARM64_REG_V0,
    UC_ARM64_REG_V1,
    UC_ARM64_REG_V2,
    UC_ARM64_REG_V3,
    UC_ARM64_REG_V4,
    UC_ARM64_REG_V5,
    UC_ARM64_REG_V6,
    UC_ARM64_REG_V7,
    UC_ARM64_REG_V8,
    UC_ARM64_REG_V9,
    UC_ARM64_REG_V10,
    UC_ARM64_REG_V11,
    UC_ARM64_REG_V12,
    UC_ARM64_REG_V13,
    UC_ARM64_REG_V14,
    UC_ARM64_REG_V15,
    UC_ARM64_REG_V16,
    UC_ARM64_REG_V17,
    UC_ARM64_REG_V18,
    UC_ARM64_REG_V19,
    UC_ARM64_REG_V20,
    UC_ARM64_REG_V21,
    UC_ARM64_REG_V22,
    UC_ARM64_REG_V23,
    UC_ARM64_REG_V24,
    UC_ARM64_REG_V25,
    UC_ARM64_REG_V26,
    UC_ARM64_REG_V27,
    UC_ARM64_REG_V28,
    UC_ARM64_REG_V29,
    UC_ARM64_REG_V30,
    UC_ARM64_REG_V31,

    //> pseudo registers
    UC_ARM64_REG_PC, // program counter register

    UC_ARM64_REG_CPACR_EL1,

    //> thread registers, depreciated, use UC_ARM64_REG_CP_REG instead
    UC_ARM64_REG_TPIDR_EL0,
    UC_ARM64_REG_TPIDRRO_EL0,
    UC_ARM64_REG_TPIDR_EL1,

    UC_ARM64_REG_PSTATE,

    //> exception link registers, depreciated, use UC_ARM64_REG_CP_REG instead
    UC_ARM64_REG_ELR_EL0,
    UC_ARM64_REG_ELR_EL1,
    UC_ARM64_REG_ELR_EL2,
```

```
    UC_ARM64_REG_ELR_EL3,

    //> stack pointers registers, depreciated, use UC_ARM64_REG_CP_REG instead
    UC_ARM64_REG_SP_EL0,
    UC_ARM64_REG_SP_EL1,
    UC_ARM64_REG_SP_EL2,
    UC_ARM64_REG_SP_EL3,

    //> other CP15 registers, depreciated, use UC_ARM64_REG_CP_REG instead
    UC_ARM64_REG_TTBR0_EL1,
    UC_ARM64_REG_TTBR1_EL1,

    UC_ARM64_REG_ESR_EL0,
    UC_ARM64_REG_ESR_EL1,
    UC_ARM64_REG_ESR_EL2,
    UC_ARM64_REG_ESR_EL3,

    UC_ARM64_REG_FAR_EL0,
    UC_ARM64_REG_FAR_EL1,
    UC_ARM64_REG_FAR_EL2,
    UC_ARM64_REG_FAR_EL3,

    UC_ARM64_REG_PAR_EL1,

    UC_ARM64_REG_MAIR_EL1,

    UC_ARM64_REG_VBAR_EL0,
    UC_ARM64_REG_VBAR_EL1,
    UC_ARM64_REG_VBAR_EL2,
    UC_ARM64_REG_VBAR_EL3,

    UC_ARM64_REG_CP_REG,

    //> floating point control and status registers
    UC_ARM64_REG_FPCR,
    UC_ARM64_REG_FPSR,

    UC_ARM64_REG_ENDING, // <-- mark the end of the list of registers

    //> alias registers

    UC_ARM64_REG_IP0 = UC_ARM64_REG_X16,
    UC_ARM64_REG_IP1 = UC_ARM64_REG_X17,
    UC_ARM64_REG_FP = UC_ARM64_REG_X29,
    UC_ARM64_REG_LR = UC_ARM64_REG_X30,
} uc_arm64_reg;
```

arm64指令（支持hook的指令）

```
//> ARM64 instructions
typedef enum uc_arm64_insn {
    UC_ARM64_INS_INVALID = 0,

    UC_ARM64_INS_MRS,
```

```
    UC_ARM64_INS_MSR,
    UC_ARM64_INS_SYS,
    UC_ARM64_INS_SYSL,

    UC_ARM64_INS_ENDING
} uc_arm64_insn;
```

Unicorn相关错误

- unicorn - 简书 (jianshu.com)

```
typedef enum uc_err {
    UC_ERR_OK = 0,   // 无错误
    UC_ERR_NOMEM,       // 内存不足: uc_open(), uc_emulate()
    UC_ERR_ARCH,      // 不支持的架构: uc_open()
    UC_ERR_HANDLE,    // 不可用句柄
    UC_ERR_MODE,      // 不可用/不支持架构: uc_open()
    UC_ERR_VERSION,   // 不支持版本 (中间件)
    UC_ERR_READ_UNMAPPED, // 由于在未映射的内存上读取而退出模拟: uc_emu_start()
    UC_ERR_WRITE_UNMAPPED, // 由于在未映射的内存上写入而退出模拟: uc_emu_start()
    UC_ERR_FETCH_UNMAPPED, // 由于在未映射的内存中获取数据而退出模拟: uc_emu_start()
    UC_ERR_HOOK,     // 无效的hook类型: uc_hook_add()
    UC_ERR_INSN_INVALID, // 由于指令无效而退出模拟: uc_emu_start()
    UC_ERR_MAP, // 无效的内存映射: uc_mem_map()
    UC_ERR_WRITE_PROT, // 由于UC_MEM_WRITE_PROT冲突而停止模拟: uc_emu_start()
    UC_ERR_READ_PROT, // 由于UC_MEM_READ_PROT冲突而停止模拟: uc_emu_start()
    UC_ERR_FETCH_PROT, // 由于UC_MEM_FETCH_PROT冲突而停止模拟: uc_emu_start()
    UC_ERR_ARG,      // 提供给uc_xxx函数的无效参数
    UC_ERR_READ_UNALIGNED,   // 未对齐读取
    UC_ERR_WRITE_UNALIGNED,   // 未对齐写入
    UC_ERR_FETCH_UNALIGNED,   // 未对齐的提取
    UC_ERR_HOOK_EXIST,   // 此事件的钩子已经存在
    UC_ERR_RESOURCE,     // 资源不足: uc_emu_start()
    UC_ERR_EXCEPTION, // 未处理的CPU异常
    UC_ERR_TIMEOUT // 模拟超时
} uc_err;
```

- https://rev.ng/gitlab/angr/unicorn/raw/dca32a875e14e35403c62c82c7c15f46c5ef450c/uc.c

```
UNICORN_EXPORT
const char *uc_strerror(uc_err code)
{
    switch(code) {
        default:
            return "Unknown error code";
        case UC_ERR_OK
            return "OK (UC_ERR_OK)";
        case UC_ERR_NOMEM
            return "No memory available or memory not present (UC_ERR_NOMEM)";
        case UC_ERR_ARCH
            return "Invalid/unsupported architecture (UC_ERR_ARCH)";
        case UC_ERR_HANDLE
            return "Invalid handle (UC_ERR_HANDLE)";
```

```
        case UC_ERR_MODE
            return "Invalid mode (UC_ERR_MODE)";
        case UC_ERR_VERSION
            return "Different API version between core & binding (UC_ERR_VERSION)";
        case UC_ERR_READ_UNMAPPED
            return "Invalid memory read (UC_ERR_READ_UNMAPPED)";
        case UC_ERR_WRITE_UNMAPPED
            return "Invalid memory write (UC_ERR_WRITE_UNMAPPED)";
        case UC_ERR_FETCH_UNMAPPED
            return "Invalid memory fetch (UC_ERR_FETCH_UNMAPPED)";
        case UC_ERR_HOOK
            return "Invalid hook type (UC_ERR_HOOK)";
        case UC_ERR_INSN_INVALID
            return "Invalid instruction (UC_ERR_INSN_INVALID)";
        case UC_ERR_MAP
            return "Invalid memory mapping (UC_ERR_MAP)";
        case UC_ERR_WRITE_PROT
            return "Write to write-protected memory (UC_ERR_WRITE_PROT)";
        case UC_ERR_READ_PROT
            return "Read from non-readable memory (UC_ERR_READ_PROT)";
        case UC_ERR_FETCH_PROT
            return "Fetch from non-executable memory (UC_ERR_FETCH_PROT)";
        case UC_ERR_ARG
            return "Invalid argument (UC_ERR_ARG)";
        case UC_ERR_READ_UNALIGNED
            return "Read from unaligned memory (UC_ERR_READ_UNALIGNED)";
        case UC_ERR_WRITE_UNALIGNED
            return "Write to unaligned memory (UC_ERR_WRITE_UNALIGNED)";
        case UC_ERR_FETCH_UNALIGNED
            return "Fetch from unaligned memory (UC_ERR_FETCH_UNALIGNED)";
        case UC_ERR_RESOURCE
            return "Insufficient resource (UC_ERR_RESOURCE)";
    }
}
```

参考资料

- 【整理】Unicorn相关内容
- 【整理】Unicorn相关文档和资料
- 【已解决】iOS逆向：unicorn的基本用法和基本逻辑
- 【记录】Unicorn相关知识：Unicorn和QEMU
- 【已解决】iOS逆向：Mac中安装unicorn
- 【已解决】Mac M2 Max中安装Unicorn
- 【已解决】Mac M2 Max中安装cmake
- 【已解决】Mac中安装pkg-config
- 【记录】Mac中用VSCode调试unicorn的示例代码sample_arm64.py
- 【已解决】Unicorn模拟ARM代码：优化内存分配布局内存映射
- 【已解决】Unicorn模拟arm64函数：___lldb_unnamed_symbol2575$$akd优化内存代码布局和输出日志
- 【已解决】iPhone11中ARM汇编Stack栈指针SP的增长方向
- 【已解决】ARM中汇编字节序的大端和小端
- 【未解决】iOS逆向：unicorn中传递函数指针参数
- 【已解决】Unicorn模拟arm64：PC在+4404时报错UC_ERR_MAP
- 【已解决】Unicorn模拟ARM代码：优化给内存地址写入对应的值
- 【已解决】unicorn模拟ARM中LR和SP寄存器堆栈信息
- 【已解决】unicorn中没有触发后续代码的hook函数hook_code
- 【未解决】unicorn中用UC_HOOK_INSN去给指令加上hook
- 【已解决】unicorn中给内存的读和写单独加上hook以辅助调试异常情况
- 【已解决】unicorn模拟ARM代码：分析内存读取和写入分析代码模拟逻辑
- 【已解决】Unicorn模拟ARM代码：优化hook打印逻辑
- 【已解决】Unicorn模拟ARM汇编：优化hook_code调试打印指令的输出日志
- 【已解决】Unicorn模拟ARM代码：优化log调试打印寄存器值
- 【已解决】Unicorn中hook时当特定位置代码时查看打印寄存器的值
- 【规避解决】Unicorn模拟ARM：去hook查看将要解码的opcode二进制
- 【已解决】unicorn中hook_code中查看当前要执行的二进制code代码指令数据
- 【已解决】iOS逆向：unicorn查看当前被识别出是什么ARM汇编指令
- 【已解决】Unicorn模拟arm64e代码时把mov识别成movz
- 【已解决】Unicorn模拟arm64：判断遇到指令ret时结束停止模拟
- 【未解决】unicorn模拟ARM汇编如何忽略特定指令为nop空指令
- 【未解决】Unicorn模拟arm：函数___lldb_unnamed_symbol2575$$akd模拟完毕但是没有生成要的结果
- 【未解决】iOS逆向akd：用Unicorn模拟运行arm64的akd函数sub_1000A0460的opcode代码
- 【已解决】Unicorn模拟arm64代码：***lldb_unnamed_symbol2575$$akd*调用子函数**lldb_unnamed_symbol2567$$akd
- 【未解决】Unicorn模拟arm：给___lldb_unnamed_symbol2567$$akd函数准备调试环境
- 【未解决】研究对比___lldb_unnamed_symbol2575$$akd的+4448到+8516的逻辑的真实过程和模拟过程之间的差异
- 【已解决】iOS逆向akd之Unicorn模拟arm64代码：PC在0x64378时出错Memory UNMAPPED at 0x0
- 【已解决】Unicorn模拟arm64：PC在+7428时blr x8报错UC_ERR_MAP

- 【已解决】Unicorn模拟arm64：PC在+9296时报错UC_ERR_MAP 0xFFFFFFFFFFFFFFFE
- 【已解决】Unicorn模拟arm64代码：PC在+552时报错ERROR Invalid memory mapping
- 【已解决】Unicorn模拟arm64：PC在+4448时报错UC_ERR_MAP at 0xFFFFFFFFFFFFFFFDUC_ERR_MAP
- 【已解决】Unicorn模拟arm64：PC在+7396时UC_ERR_MAP出错0xFFFFFFFFFFFFFFFE
- 【已解决】iOS逆向akd用Unicorn模拟代码：PC在0x000100CC时报错Invalid memory mapping UC_ERR_MAP 0xFFFFFFFFFFFFFFFE
- 【已解决】iOS逆向akd用Unicorn模拟ARM：PC在0x0001011C时出错Invalid memory mapping UC_ERR_MAP
- 【已解决】Unicorn模拟arm64：PC在+4404时报错UC_ERR_MAP
- 【已解决】iOS逆向akd之Unicorn模拟arm64代码：PC在0x00010088时出错br跳转0地址
- 【已解决】unicorn模拟ARM64代码报错：ERROR Invalid memory mapping UC_ERR_MAP
- 【已解决】Unicorn模拟ARM代码出错：Memory UNMAPPED at 0x24C6
- 【未解决】Unicorn模拟ARM代码出错：PC是0x10090时Memory UNMAPPED at 0x100AF6C88
- 【已解决】Unicorn模拟arm64代码：尝试批量一次性解决全部的br跳转导致内存映射错误UC_ERR_MAP
- 【已解决】Unicorn模拟arm64代码：搞懂___lldb_unnamed_symbol2575$$akd函数br跳转涉及到的固定的值的内存范围
- 【已解决】Unicorn模拟arm64代码：计算br跳转涉及到的x10函数偏移量地址的合适的范围
- 【已解决】Unicorn模拟arm64代码：计算br跳转涉及到的x9的合适的范围
- 【已解决】Unicorn模拟arm64代码：导出lldb调试时x9和x10两个段的实际数据值到文件
- 【已解决】Unicorn模拟arm64：修正导出的x10的带偏移量的函数地址
- 【已解决】Unicorn模拟arm64代码：把导出的x9和x10的2段数据导入到Unicorn模拟代码中
- 【已解决】unicorn代码报错：ERROR Invalid memory write UC_ERR_WRITE_UNMAPPED
- 【已解决】unicorn模拟ARM64代码：给UC_ERR_WRITE_UNMAPPED单独加上hook看出错时详情
- 【已解决】Unicorn模拟ARM64代码：手动把braa改为br指令看是否还会报错UC_ERR_EXCEPTION
- 【已解决】通过修改ARM汇编二进制文件实现Unicorn忽略执行特定指令
- 【已解决】Python中把byteaddary转成64位的unsigned long long的数值
- 【已解决】Python中把int值转换成字节bytes
- 【已解决】unicorn代码mem_read报错：mem_read() missing 1 required positional argument size
- 【已解决】Unicorn模拟ARM代码：写入正确的地址但是读取出来数据值仍旧是错的
- 【已解决】Unicorn模拟ARM代码：mem_read内存读取的值和ldr指令加载出来的值不一样
- 【已解决】Unicorn模拟ARM：用内存映射并写入地址0x75784的内容
- 【已解决】Unicorn中Python中的ARM64的CPSR寄存器定义
- 【未解决】unicorn如何模拟ARM中PAC指令pacibsp
- 【未解决】iOS逆向：用unicorn模拟执行arm64e的arm汇编代码
- 【未解决】Unicorn模拟ARM代码报错：ERROR Unhandled CPU exception UC_ERR_EXCEPTION
- 【已解决】Unicorn模拟ARM64的函数中调用其他子函数
- 【未解决】Unicorn模拟arm：确保子函数___lldb_unnamed_symbol2567$$akd参数值正确
- 【已解决】Unicorn模拟arm64代码：模拟___lldb_unnamed_symbol2567$$akd直接返回值
- 【已解决】Unicorn模拟ARM64：模拟造出返回特定值的空函数的arm汇编代码的opcode
- 【已解决】Unicorn模拟ARM64代码：参考afl-unicorn的UnicornSimpleHeap模拟malloc
- 【已解决】Unicorn模拟ARM64：如何模拟malloc分配内存
- 【已解决】Unicorn模拟ARM：模拟malloc报错内存重叠Heap over underflow
- 【已解决】Unicorn模拟malloc内存分配：返回重复内存地址
- 【已解决】Unicorn模拟arm64：模拟free释放内存

- 【已解决】Unicorn模拟arm64：模拟vm_deallocate释放内存
- 【未解决】iOS逆向：如何反代码混淆反混淆去混淆
-
- 反汇编利器：Capstone
-
- unicorn - 简书 (jianshu.com)
- Unicorn.js: ARM (alexaltea.github.io)
- iOS Tampering and Reverse Engineering - OWASP Mobile Application Security
- Unicorn & QEMU – Unicorn – The Ultimate CPU emulator (unicorn-engine.org)
- 为什么使用汇编可以 Hook objc_msgSend（上）- 汇编基础 - 一片瓜田 (desgard.com)
- Opcode - Wikipedia
- [原创] Unicorn 在 Android 的应用-『Android安全』-看雪安全论坛
- Programming with Python language – Capstone – The Ultimate Disassembler (capstone-engine.org)
- capstone/bindings/python at master · capstone-engine/capstone · GitHub
- Unicorn快速入门 - iPlayForSG - 博客园 (cnblogs.com)
- Online ARM to HEX Converter (armconverter.com)
- Showcases – Unicorn – The Ultimate CPU emulator
- afl-unicorn: Fuzzing Arbitrary Binary Code | by Nathan Voss | HackerNoon.com | Medium
- afl-unicorn/unicorn_loader.py at master · Battelle/afl-unicorn · GitHub
-