lab1：Rev基础

3250101697冯梓航　　实验环境：Ubuntu 24.04 (aarch64), GCC 13.3.0, Clang 18.1.3

---

1. Task 1：rev_begin (57%)

1.1 Compilation (24%)

1.1.1 编译工具

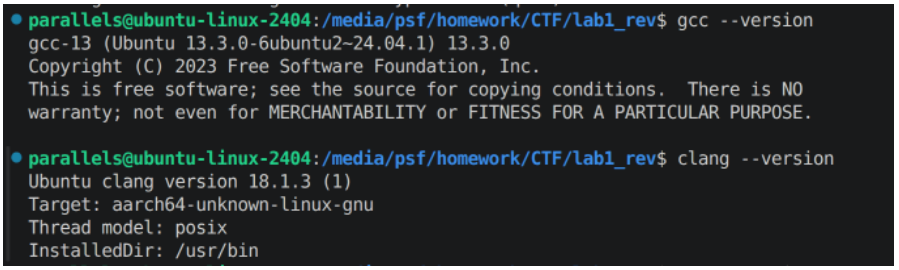

Image 1

安装过程中曾遇到 libllvm18/libclang 版本冲突（已安装 `-1ubuntu1` 版本而仓库中为 `-1` 版本），通过 `--allow-downgrades` 降级后成功安装。

1.1.2 使用 g++ 对代码进行预处理（宏展开）

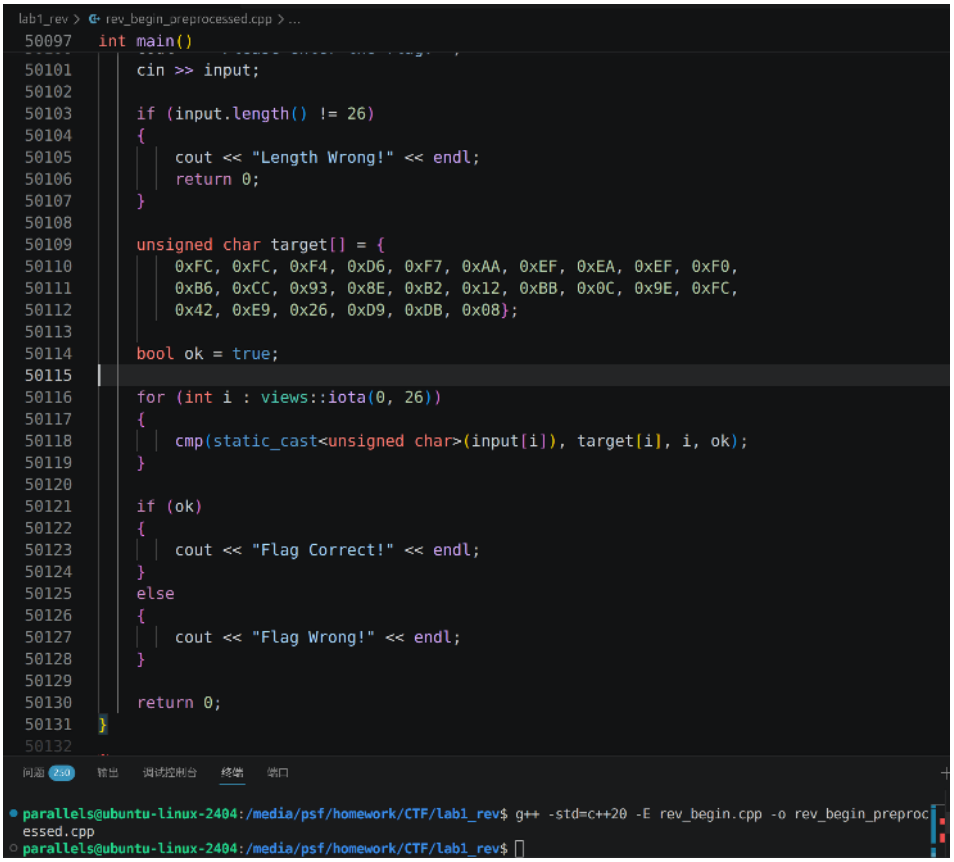

Image 2

展开所有 `#include` 头文件、宏定义，去除注释
展开后文件末尾保留原始代码逻辑

1.1.3 使用 g++ 将代码编译为汇编代码

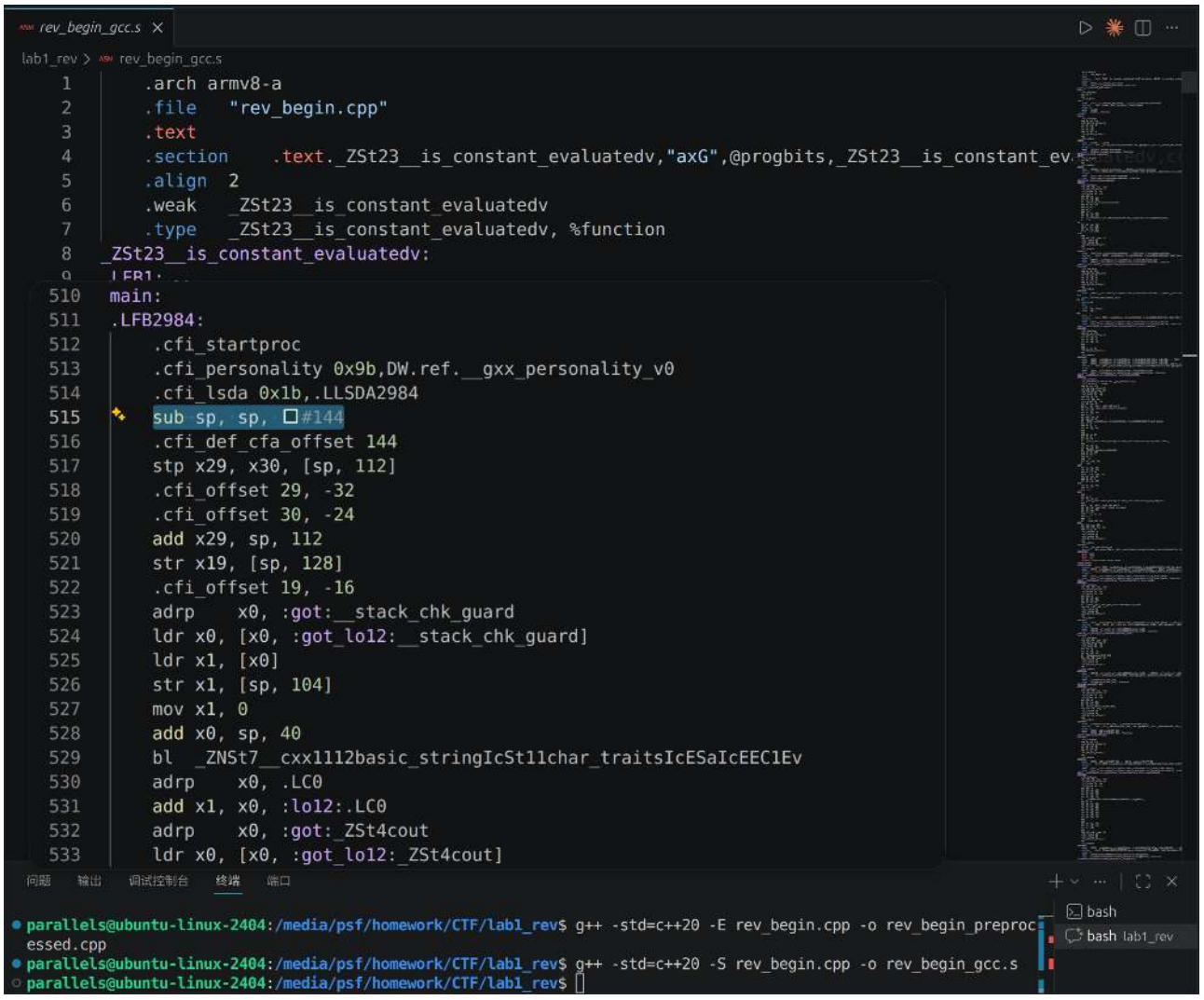

Image 3

将 C++ 代码编译为 ARM aarch64 汇编代码 (电脑是 arm 架构的)

汇编文件结构:

```
    .arch armv8-a
    .file   "rev_begin.cpp"
    .text
    // ... 各种 symbol 定义 ...

main:
.LFB2984:
    .cfi_startproc
    sub     sp, sp, #144          ; 分配栈帧 144 字节
    stp     x29, x30, [sp, 112] ; 保存帧指针和返回地址
    add     x29, sp, 112
    // ... 调用 cout << "Please enter the flag: "
    // ... 调用 cin >> input
```

```
14        // ... 长度检查、循环变换验证等
```

1.1.4 使用 g++ 将代码汇编（不链接）为目标文件

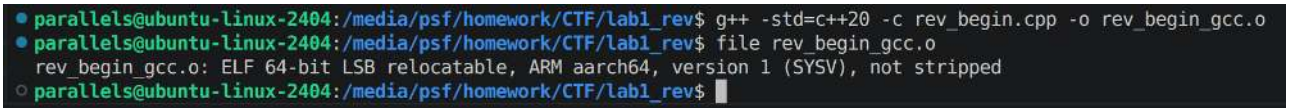

Image 4

将汇编代码翻译为机器码（目标文件），但不进行链接。

可以看到它是 **relocatable**（可重定位）文件，尚未链接。

1.1.5 使用 g++ 进行链接，生成可执行文件，并尝试运行

Image 5

1.1.6 使用 clang++ 生成文本格式的 LLVM IR

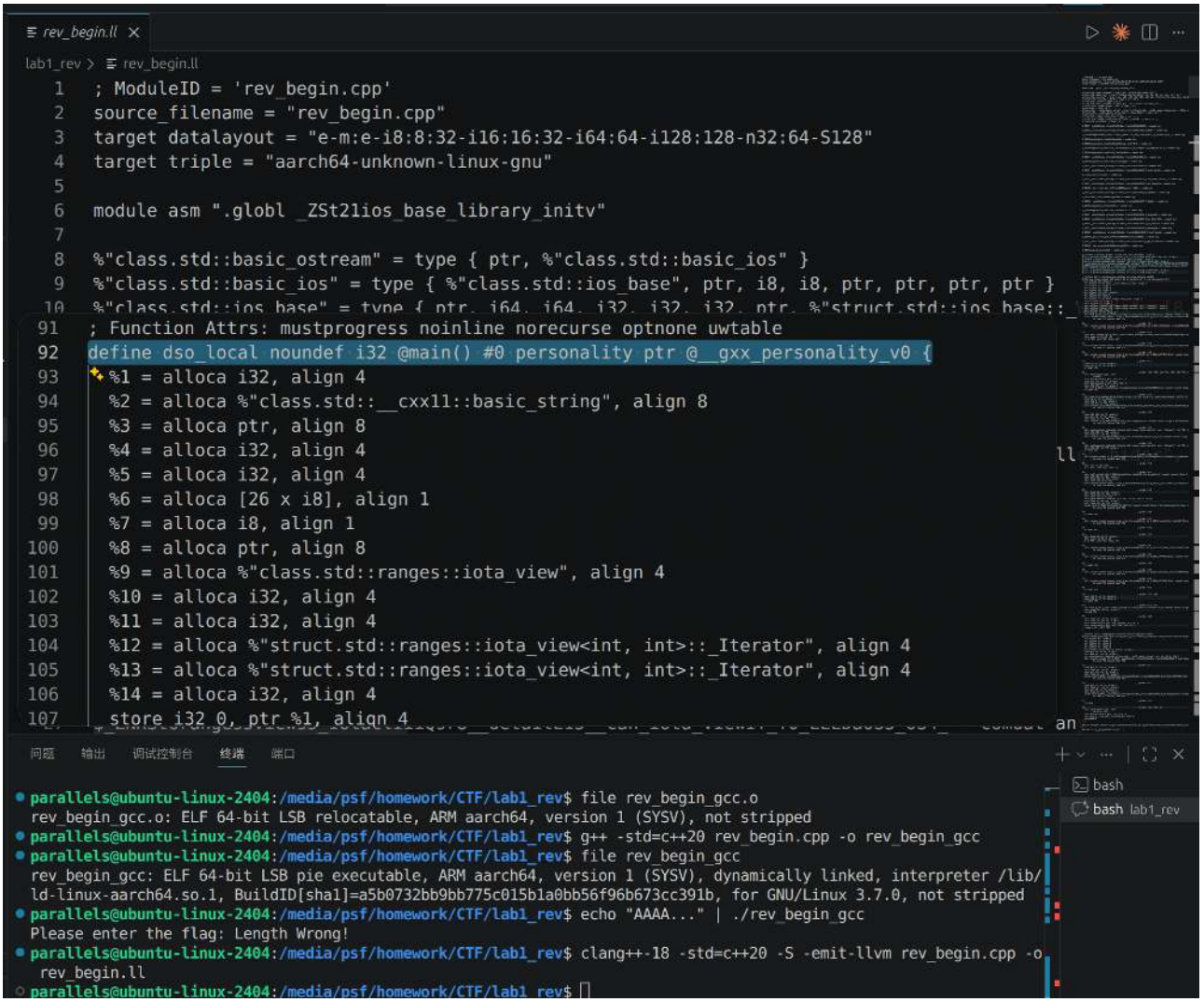

Image 6

生成人类可读的 LLVM 中间表示（IR），这是 LLVM 编译器的核心中间语言。

LLVM IR 使用 SSA（静态单赋值）形式，以 `%` 编号的虚拟寄存器承载所有中间值。

1.1.7 使用 clang++ 生成二进制形式的 LLVM Bitcode

Image 7

生成二进制格式的 LLVM Bitcode，可供后续 LTO（链接时优化）等使用。

1.1.8 使用 clang++ 生成可执行文件，并尝试运行

Image 8

使用 clang++ 完整编译生成可执行文件。

运行结果与 g++ 编译版本行为一致。

1.2 Analyze and Tool Use (33%)

1.2.1 1. 逆推 Flag

根据源码中的变换逻辑逆推 flag。`cmp` lambda 对每个字符的变换为：

```
c = a ^ 0x42          // XOR 0x42
c = c + i             // 加下标
c = ~c                // 按位取反
c = c ^ ((i*7) & 0xFF)  // XOR (i*7) 低8位
```

逆向推导（逆序执行逆操作）：

```
target = [0xFC, 0xFC, 0xF4, 0xD6, 0xF7, 0xAA, 0xEF, 0xEA, 0xEF, 0xF0,
          0xB6, 0xCC, 0x93, 0x8E, 0xB2, 0x12, 0xBB, 0x0C, 0x9E, 0xFC,
          0x42, 0xE9, 0x26, 0xD9, 0xDB, 0x08]

for i, b in enumerate(target):
    c = b ^ ((i * 7) & 0xFF)    # 逆 XOR
    c = (~c) & 0xFF              # 逆 NOT（注意 8-bit 掩码）
    c = (c - i) & 0xFF           # 逆加法
    a = c ^ 0x42                 # 逆 XOR
    print(chr(a), end='')
```

得到 flag: **`AAA{R3v_beG1n_c7f1O1_2o26}`**

验证：

```
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$  echo "AAA{R3v_beG1n_c7f101_2o26}" |
./rev_begin_gcc && echo "AAA{R3v_beG1n_c7f101_2o26}" | ./rev_begin_clang
Please enter the flag: Flag Correct!
Please enter the flag: Flag Correct!
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$
```

Image 9

1.2.2 2. 静态分析

使用 IDA Pro 分别对 GCC 和 Clang 编译的两个可执行文件进行静态反编译分析。

1.2.2.1 2.1 `std::views::iota` 的处理对比

GCC — 直接在 main 中构造 `iota_view{0, 26}`，无中间层：

```
if ( std::string::length(v13) == 26 )
{
  *(_OWORD *)v14 = xmmword_1DD0;
  *(_OWORD *)&v14[10] = *(__int128 *)((char *)&xmmword_1DD0 + 10);
  v7 = 1;
  v11 = 0x1A00000000LL;
  v12 = &v11;
  v8 = std::ranges::iota_view<int,int>::begin(&v11);
  v9 = std::ranges::iota_view<int,int>::end(v12);
  while ( (((unsigned __int8)std::ranges::operator==(&v8, &v9) ^ 1) & 1) != 0 )
  {
    v10 = std::ranges::iota_view<int,int>::_Iterator::operator*(&v8);
    v4 = (unsigned __int8 *)std::string::operator[](v13, (int)v10);
    ZNK3cmpMUlTyT_S0_iRbE_clIhEEDaS0_S0_iS1_(&cmp, *v4, (unsigned __int8)v14[v10], v10, &v7);
    std::ranges::iota_view<int,int>::_Iterator::operator++(&v8);
  }
  if ( (v7 & 1) != 0 )
    v5 = std::operator<<<std::char_traits<char>>(&std::cout, "Flag Correct!");
  else
    v5 = std::operator<<<std::char_traits<char>>(&std::cout, "Flag Wrong!");
  std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
}
```

GCC

GCC：直接构造 iota_view

Lambda独立函数调用

Image 10

- `0x1A00000000LL`：将两个 `int`（0 和 26）合并为一次 64-bit 写入
- 直接调用 `iota_view::begin()` / `end()`，走标准迭代器模型

Clang — 多了 `views::_Iota` CPO 这一层：

```
24  if ( std::string::length(v18) == 26 )
25  {
26    *(_OWORD *)v16 = xmmword_1A7C;
27    *(_OWORD *)&v16[10] = *(__int128 *)((char *)&xmmword_1A7C + 10);
28    v15 = 1;
29    v12 = 0;
30    v11 = 26;
31    can_iota_viewIT_T0_EEEDaOS3_OS4 = ZNKSt6ranges5views5_IotaclIiiQsr8__detailE15__can_iota_viewIT_T0_EEEDaOS3_OS4_(
32                                        &std::ranges::views::iota,
33                                        &v12,
34                                        &v11);
35    p_can_iota_viewIT_T0_EEEDaOS3_OS4 = &can_iota_viewIT_T0_EEEDaOS3_OS4;
36    v10 = std::ranges::iota_view<int,int>::begin(&can_iota_viewIT_T0_EEEDaOS3_OS4);
37    v9 = std::ranges::iota_view<int,int>::end(p_can_iota_viewIT_T0_EEEDaOS3_OS4);
38    while ( (ZNSt6ranges9iota_viewIiiE9_IteratorFeqERKS2_S4_Q19equality_comparableIT_E(&v10, &v9) & 1) == 0 )
39    {
40      v8 = std::ranges::iota_view<int,int>::_Iterator::operator*(&v10);
41      v6 = (unsigned __int8 *)std::string::operator[](v18, (int)v8);
42      $_0::operator()<unsigned char>(&cmp, *v6, (unsigned __int8)v16[v8], v8, &v15);
43      std::ranges::iota_view<int,int>::_Iterator::operator++(&v10);
44    }
45    if ( (v15 & 1) != 0 )
46    {
47      v5 = std::operator<<<std::char_traits<char>>(&std::cout, "Flag Correct!");
48      std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
49    }
50    else
```

Clang

Clang：通过 views::_Iota CPO 间接

Image 11

- 先调 `views::_Iota::operator()`（CPO），内部校验类型后才构造 `iota_view`

- `end()` 返回 **sentinel** 类型，循环用 `IteratorFeq(Iterator, Sentinel)` 比较

方面	GCC	Clang
iota 构造	直接实例化	通过 CPO 间接创建
调用层级	1 层	2 层（多 CPO）
结束判断	operator== sentinel	IteratorFeq

Table 1

1.2.2.2 2.2 Lambda 表达式的实现对比

GCC — 整个 lambda 融合为单行表达式，`i*7` 做强度削减 (具体见后文动态调试的反汇编):

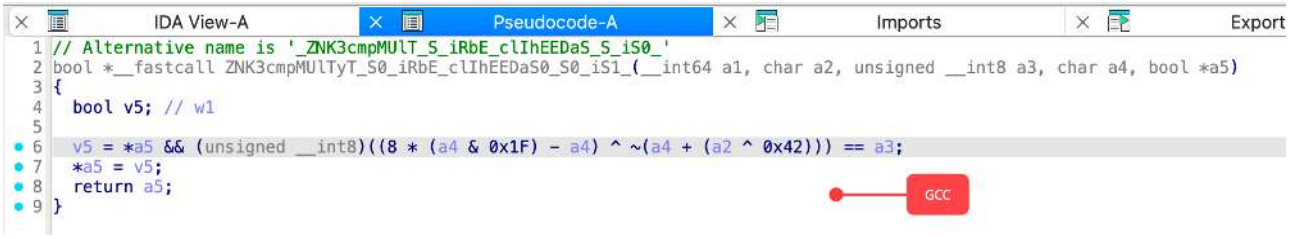

Image 12

```
v5 = *a5 && (unsigned __int8)((8 * (a4 & 0x1F) - a4) ^ ~(a4 + (a2 ^
0x42))) == a3;
```

- `8 * (a4 & 0x1F) - a4` = `i*8 - i` = `i*7`（移位替代乘法）
- `*a5 && (...)` 表达式内短路

Clang — 用 `if` 分支保持结构清晰，`i*7` 直接乘:

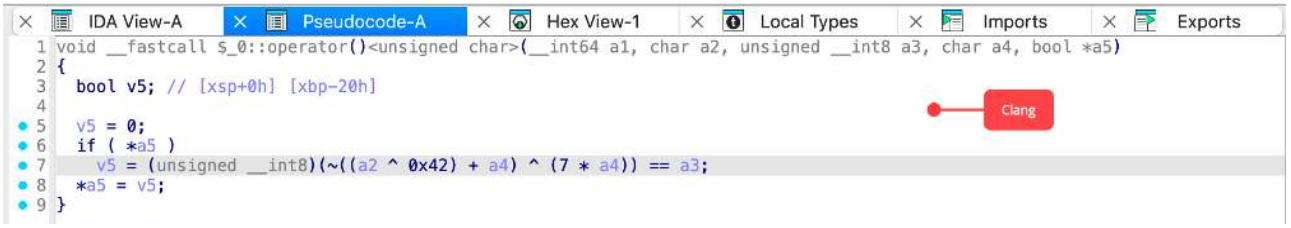

Image 13

```
v5 = 0;
if ( *a5 )
    v5 = (unsigned __int8)(~((a2 ^ 0x42) + a4) ^ (7 * a4)) == a3;
*a5 = v5;
```

- `7 * a4` 直接 `mul`，未做强度削减
- `if (*a5)` 控制流短路；闭包类型命名 `$_0`（GCC 用 `cmp`）

	GCC	Clang
结构	单行融合表达式	if 分支
i*7	i*8-i（强度削减）	7*i（直乘）
命名	cmp::{lambda}...	$_0::operator()...

Table 2

> 两者核心逻辑等价，运行结果一致。
>
> GCC 偏重算术融合与强度削减，Clang 保留更多源码结构。

1.2.3 动态调试（GDB）

1.2.3.1 进入 Lambda 内部

在 main 的 lambda 调用处（`main+308`）设断点，`stepi` 进入：

```
(gdb) break *main+308
(gdb) continue
(gdb) stepi
0x0000aaaaaaaa17e8 in cmp::{lambda}::operator()<unsigned char>()
```

`disassemble` 查看 lambda 指令：

```
0x...17e8 :  sub   sp, sp, #0x30        ← 栈帧 48 字节
...
0x...1840 :  ubfiz w0, w0, #3, #5       ← i << 3
0x...1844 :  sub   w0, w0, w1           ← i*8 - i = i*7
```

GCC 将 `i*7` 优化为 `(i&0x1F)*8 - i`，用移位+减法替代乘法，即静态分析中观察到的强度削减。

```
(gdb) break *main+308
Note: breakpoint 1 also set at pc 0xaaaaaaaa140c.
Breakpoint 2 at 0xaaaaaaaa140c
(gdb) c
Continuing.

Breakpoint 1, 0x0000aaaaaaaa140c in main ()
(gdb) stepi
0x0000aaaaaaaa17e8 in auto cmp::{lambda<typename $T0>($T0, $T0, int, bool&)#1}::operator()<unsigned
char>(unsigned char, unsigned char, int, bool&) const ()
(gdb) disassemble
Dump of assembler code for function  ZNK3cmpMUlTyT_S0_iRbE_clIhEEDaS0_S0_iS1_:
=> 0x0000aaaaaaaa17e8 <+0>:     sub     sp, sp, #0x30
   0x0000aaaaaaaa17ec <+4>:     str     x0, [sp, #24]
   0x0000aaaaaaaa17f0 <+8>:     strb    w1, [sp, #23]
   0x0000aaaaaaaa17f4 <+12>:    strb    w2, [sp, #22]
   0x0000aaaaaaaa17f8 <+16>:    str     w3, [sp, #16]
   0x0000aaaaaaaa17fc <+20>:    str     x4, [sp, #8]
   0x0000aaaaaaaa1800 <+24>:    ldrb    w1, [sp, #23]
   0x0000aaaaaaaa1804 <+28>:    mov     w0, #0x42                  // #66
   0x0000aaaaaaaa1808 <+32>:    eor     w0, w1, w0
   0x0000aaaaaaaa180c <+36>:    strb    w0, [sp, #47]
   0x0000aaaaaaaa1810 <+40>:    ldr     w0, [sp, #16]
   0x0000aaaaaaaa1814 <+44>:    and     w0, w0, #0xff
   0x0000aaaaaaaa1818 <+48>:    ldrb    w1, [sp, #47]
   0x0000aaaaaaaa181c <+52>:    add     w0, w0, w1
   0x0000aaaaaaaa1820 <+56>:    strb    w0, [sp, #47]
   0x0000aaaaaaaa1824 <+60>:    ldrb    w0, [sp, #47]
   0x0000aaaaaaaa1828 <+64>:    mvn     w0, w0
   0x0000aaaaaaaa182c <+68>:    strb    w0, [sp, #47]
   0x0000aaaaaaaa1830 <+72>:    ldr     w0, [sp, #16]
   0x0000aaaaaaaa1834 <+76>:    and     w0, w0, #0xff
   0x0000aaaaaaaa1838 <+80>:    mov     w1, w0
   0x0000aaaaaaaa183c <+84>:    mov     w0, w1
   0x0000aaaaaaaa1840 <+88>:    ubfiz   w0, w0, #3, #5
   0x0000aaaaaaaa1844 <+92>:    sub     w0, w0, w1
   0x0000aaaaaaaa1848 <+96>:    and     w0, w0, #0xff
   0x0000aaaaaaaa184c <+100>:   sxtb    w1, w0
   0x0000aaaaaaaa1850 <+104>:   ldrsb   w0, [sp, #47]
```

Image 14

1.2.3.2 3.3 正确 vs 错误输入对比

输入正确 flag： 任意循环迭代中查看 `ok` 标志：

```
(gdb) x/1xb $sp+0x7
0xffffffffe237: 0x01                    ; ok 始终为 true
```

程序输出 `Flag Correct!`

```
(gdb) break *main+308
Breakpoint 1 at 0x140c
(gdb) run < /tmp/flag.txt
Starting program: /media/psf/homework/CTF/lab1_rev/rev_begin_gcc < /tmp/flag.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Please enter the flag:
Breakpoint 1, 0x0000aaaaaaaa140c in main ()
(gdb) x/1xb $sp+0x7
0xffffffffe2b7: 0x01
(gdb) continue
Continuing.

Breakpoint 1, 0x0000aaaaaaaa140c in main ()
(gdb) x/1xb $sp+0x7
0xffffffffe2b7: 0x01
(gdb) continue
Continuing.

Breakpoint 1, 0x0000aaaaaaaa140c in main ()
(gdb) delete
Delete all breakpoints, watchpoints, tracepoints, and catchpoints? (y or n) y
(gdb) c
Continuing.
Flag Correct!
[Inferior 1 (process 318675) exited normally]
(gdb)
```

Image 15

输入错误字符串 `AAAA...`： 同样位置：

```
(gdb) x/1xb $sp+0x7
0xffffffffe237: 0x00                    ; 第四个'A'时变为 false
```

程序输出 `Flag Wrong!`

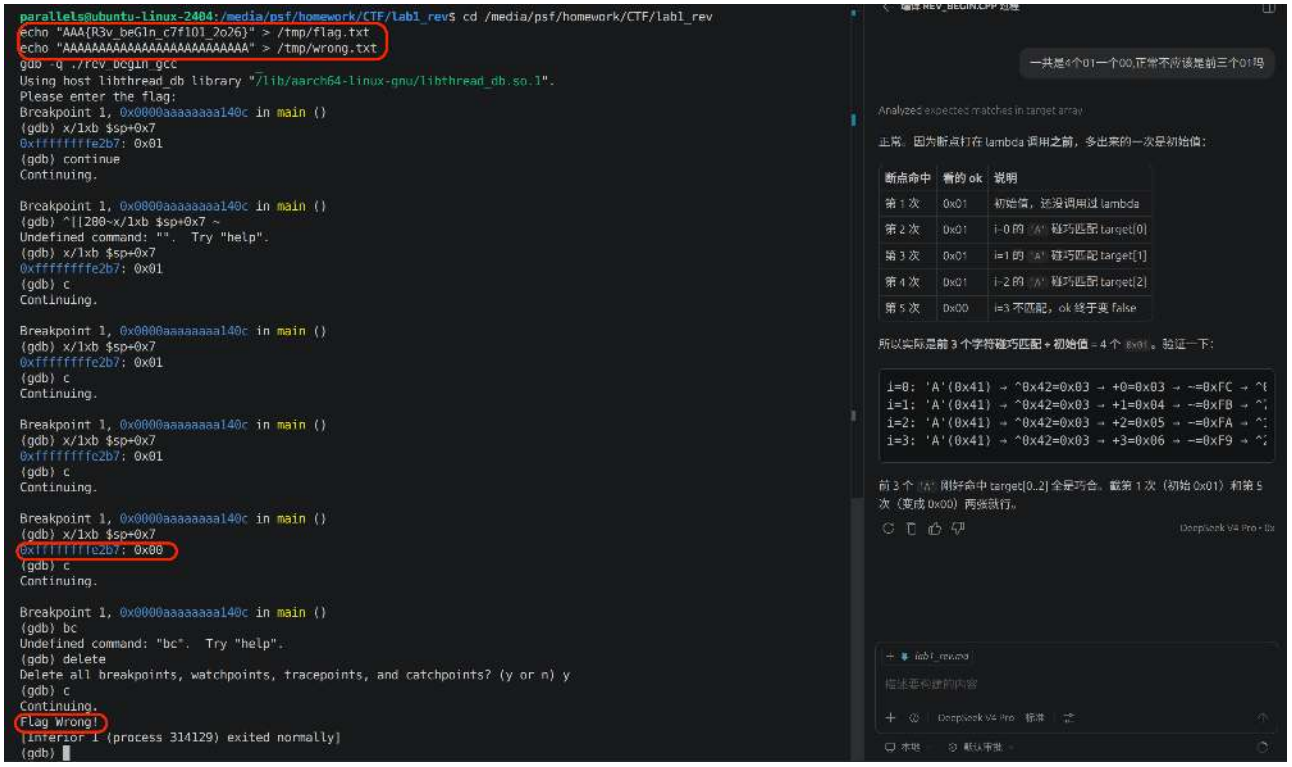

Image 16

`ok` 初始为 `0x01`，一旦某次 `cmp` 比较失败，`ok && (c==b)` 短路后 `ok` 变为 `0x00`，后续循环中即使其他字符匹配也无法恢复。

2. Task 2：rev_more (31%)

`rev_more` 是 Go 1.23 编译的 x86-64 静态链接可执行文件。

采用静态分析。

```
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$ file rev_more
rev_more: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go BuildID=hn6xmR_
mbnImEUi8h1_F/-9Y5TGzO53zYjfxp5buE/tmM9OkI6FgmyHiyish5S/WPMYVfKy0D4dKQmglAk2, BuildID[sha1]=077c20f7
ac0712eadd7d807ac6f478793c44bb7b, with debug_info, not stripped
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$
```

Image 17

2.1 逆向分析过程

2.1.1 提取字符串和符号

用 `nm` 看符号表，发现用户代码只有 `main.main` 一个函数（其余全是 Go 运行时）：

```
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$ nm rev_more | grep 'main\.'
00000000005b7fe0 D go:main.inittasks
00000000004aea60 T main.main
0000000000476be0 T runtime.main.func1
000000000044a000 T runtime.main.func2
```

Image 18

用 `strings` 搜关键字符串，定位程序行为：

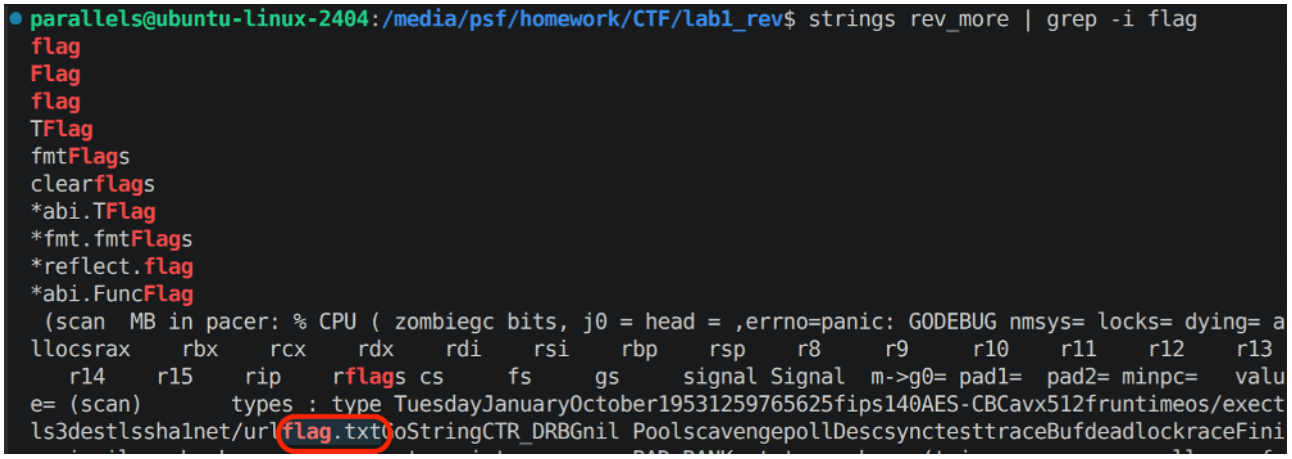

Image 19

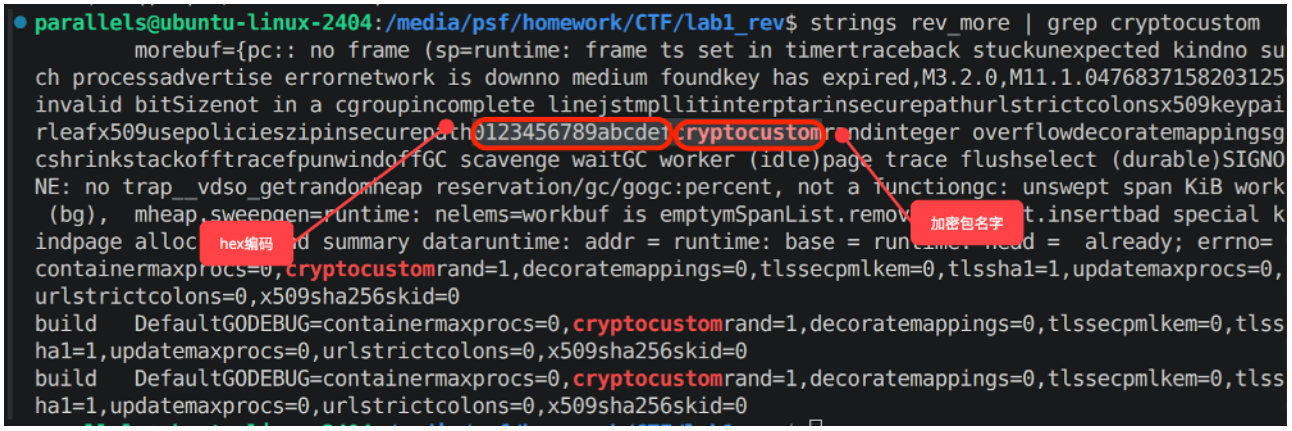

Image 20

2.1.2 安装 Go 工具链反汇编

Go 二进制保留了完整符号和行号信息，用 `go tool objdump` 可反汇编到源码行。

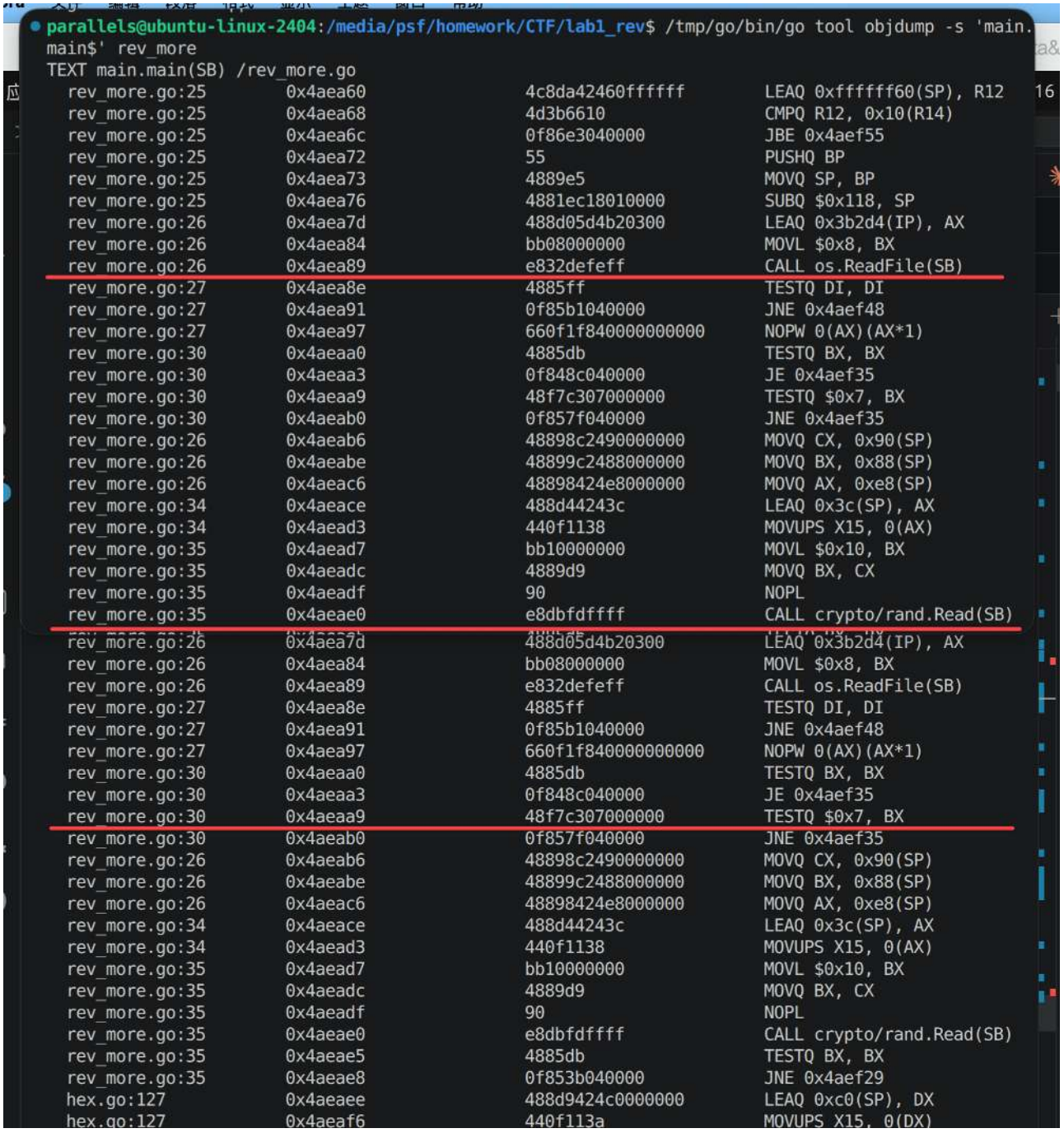

Image 21

反汇编 main.main：

```
/tmp/go/bin/go tool objdump -s 'main.main$' rev_more
```

从反汇编结果可以看出，程序先把 flag 用 XTEA 加密，再把密钥和密文 hex 输出。整个流程不比对密码，只需按加密的逆序就能解密。

flag 校验逻辑如下：

源码行	关键指令	逻辑
rev_more.go:26	CALL os.ReadFile	读取 flag.txt
rev_more.go:30	TESTQ $0x7, BX	长度必须为 8 的倍数
rev_more.go:35	CALL crypto/rand.Read	生成 16 字节随机密钥
rev_more.go:38	CALL fmt.Fprintln	hex 输出密钥（32 字符）
rev_more.go:41	BSWAP	密钥/flag 字节翻转
rev_more.go:17	CMPQ AX, $0x6f	111 轮加密循环
rev_more.go:19	LEAL 0x4a537f29	delta=0x4A537F29
rev_more.go:59	CALL fmt.Fprintln	hex 输出密文

Table 3

完整的 flag 处理流程如上表。可以看出密钥是随机的，每次运行都不一样；但加密算法是固定的，把流程反过来就是解密。编写代码，连接题目远程环境并解出 flag (5%):

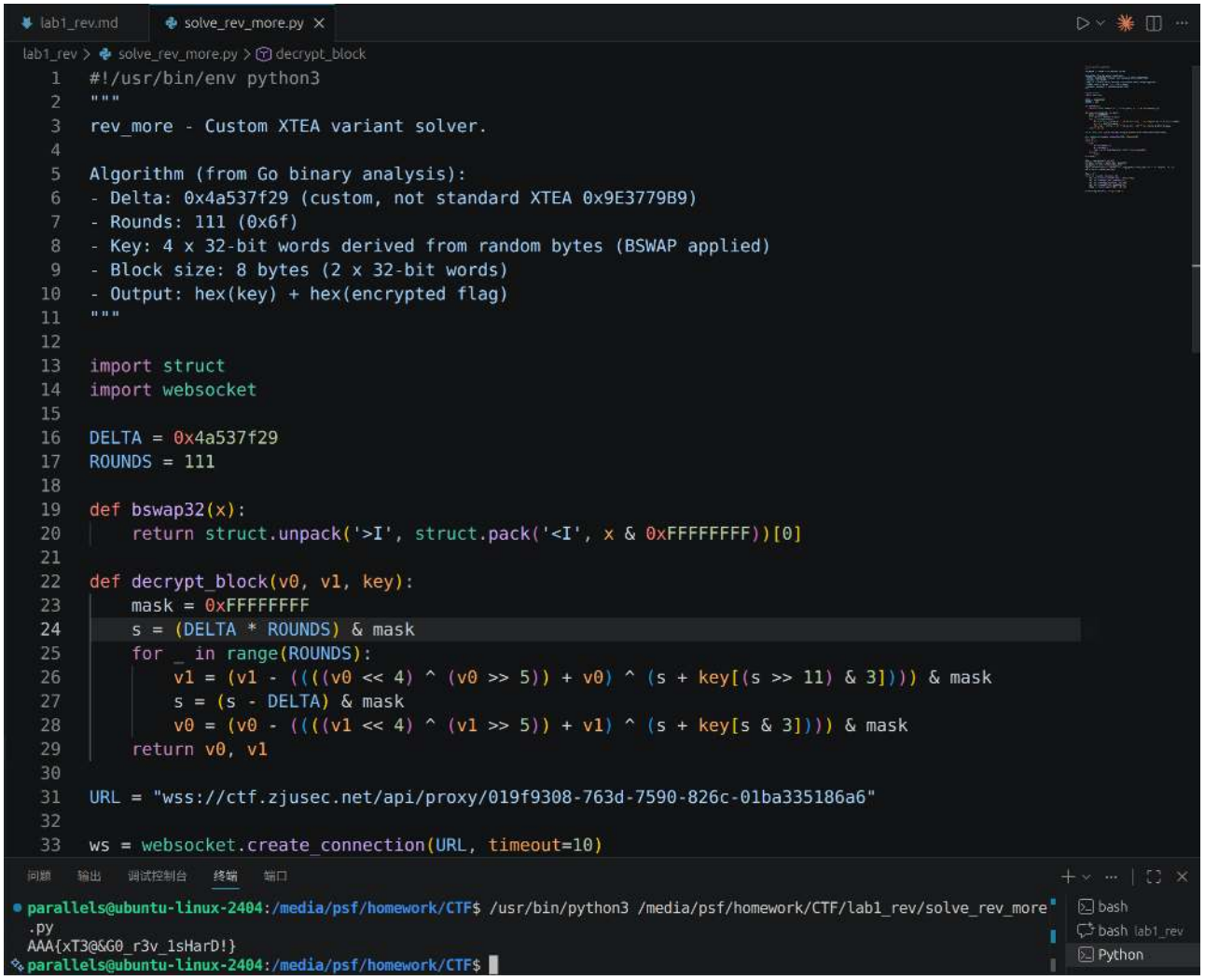

Image 22

3. Task 3：rev_again (27%)

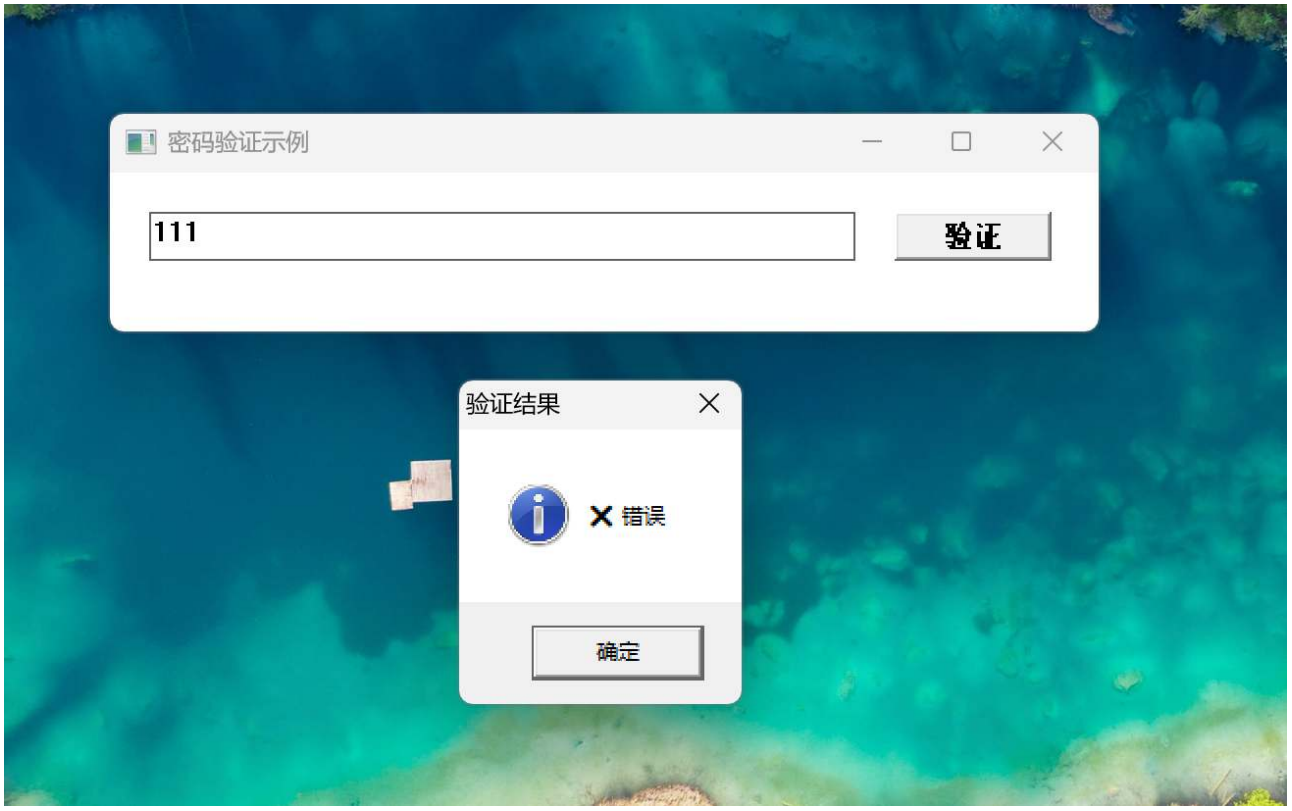

Image 23

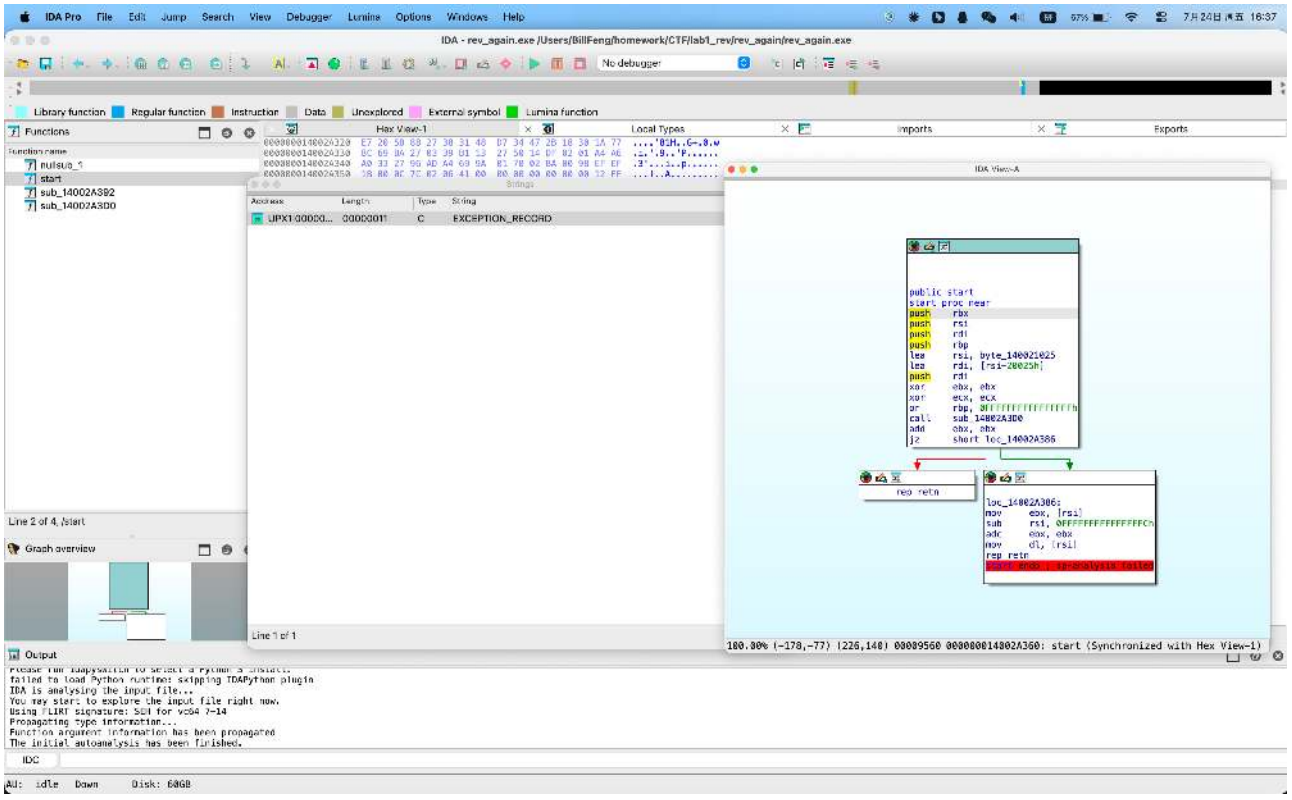

Image 24

IDA 打开后发现空空如也。判断是 UPX 壳，先脱壳：

```
/usr/bin/upx
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev$ cd /media/psf/homework/CTF/lab1_rev/rev_again && upx -
d rev_again.exe -o rev_again_unpacked.exe 2>&1
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2024
UPX 4.2.2       Markus Oberhumer, Laszlo Molnar & John Reiser    Jan 3rd 2024

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     193181 <-    121501   62.89%    win64/pe     rev_again_unpacked.exe

Unpacked 1 file.
parallels@ubuntu-linux-2404:/media/psf/homework/CTF/lab1_rev/rev_again$
```

Image 25

脱壳后看到 WinMain 创建了一个窗口，核心逻辑在 WindowProc。

```
HWND hWnd; // [rsp+358h] [rbp+2D8h]

if ( a2 == 273 )
{
  if ( (_DWORD)a3 == 1002 )
  {
    hWnd = GetDlgItem(hWndParent, 1001);
    memset(String, 0, sizeof(String));
    GetWindowTextW(hWnd, String, 256);
    v22 = &v15;
    std::wstring::basic_string<std::allocator<wchar_t>>(v8, String, &v15);
    std::__new_allocator<wchar_t>::~__new_allocator(&v15);
    wstring_to_utf8(v9, v8);
    v21 = &v16;
    std::string::basic_string<std::allocator<char>>(v10, "kEy3rd", &v16);
    std::__new_allocator<char>::~__new_allocator(&v16);
    encrypt_data(v14, v9, v10);
    v20 = &v17;
    v7[0] = &C_8_0;
    v7[1] = 44;
    std::vector<unsigned char>::vector(v13, v7, &v17);
    std::__new_allocator<unsigned char>::~__new_allocator(&v17);
    v23 = std::operator==<unsigned char,std::allocator<unsigned char>>(v14, v13);
    v19 = &v18;
    if ( v23 )
      v4 = &unk_140006028;
    else
      v4 = &unk_140006032;
    std::wstring::basic_string<std::allocator<wchar_t>>(v11, v4, &v18);
    std::__new_allocator<wchar_t>::~__new_allocator(&v18);
    v5 = (const WCHAR *)std::wstring::c_str(v11);
    MessageBoxW(hWndParent, v5, &Caption, 0x40u);
    std::wstring::~wstring(v11);
    std::vector<unsigned char>::~vector(v13);
    std::vector<unsigned char>::~vector(v14);
    std::string::~string(v10);
    std::string::~string(v9);
    std::wstring::~wstring(v8);
  }
}
else
{
  if ( a2 > 0x111 )
```

Image 26

WindowProc 的流程：按钮点击 → 读输入 → `encrypt_data(input, key="kEy3rd")` → 与硬编码的 `C_8_0` 比较。

```
__int64 __fastcall encrypt_data(__int64 a1, __int64 a2, __int64 a3)
{
  int v3; // ebx
  const unsigned __int8 *v4; // rax
  __int64 v5; // rax
  unsigned __int8 *v6; // rbx
  int v7; // esi
  const unsigned __int8 *v8; // rax
  unsigned __int8 v10[263]; // [rsp+20h] [rbp-60h] BYREF
  char v11; // [rsp+127h] [rbp+A7h] BYREF
  char *v12; // [rsp+128h] [rbp+A8h]

  v3 = std::string::size(a3);
  v4 = (const unsigned __int8 *)std::string::c_str(a3);
  cipher_init(v10, v4, v3);
  v12 = &v11;
  v5 = std::string::size(a2);
  std::vector<unsigned char>::vector(a1, v5, &v11);
  std::__new_allocator<unsigned char>::~__new_allocator(&v11);
  v6 = (unsigned __int8 *)std::vector<unsigned char>::data(a1);
  v7 = std::string::size(a2);
  v8 = (const unsigned __int8 *)std::string::c_str(a2);
  cipher_crypt(v10, v8, v7, v6);
  return a1;
}
```

Image 27

`encrypt_data` 内部调用了 `cipher_init` 和 `cipher_crypt`：

- `cipher_init` 是标准 RC4 KSA
- `cipher_crypt` 是 修改版 RC4 `j = (j + S[i] + 1) & 0xFF`

```
1  unsigned __int8 *__fastcall cipher_init(unsigned __int8 *a1, const unsigned __int8 *a2, int a3)
2  {
3    unsigned __int8 *result; // rax
4    unsigned __int8 v4; // [rsp+3h] [rbp-Dh]
5    int j; // [rsp+4h] [rbp-Ch]
6    int v6; // [rsp+8h] [rbp-8h]
7    int i; // [rsp+Ch] [rbp-4h]
8
9    for ( i = 0; i <= 255; ++i )
10   {
11     result = &a1[i];
12     *result = i;
13   }
14   LOBYTE(v6) = 0;
15   for ( j = 0; j <= 255; ++j )
16   {
17     v6 = (unsigned __int8)(a1[j] + v6 + a2[j % a3]);
18     v4 = a1[j];
19     a1[j] = a1[v6];
20     result = (unsigned __int8 *)v4;
21     a1[v6] = v4;
22   }
23   return result;
24 }
```

Image 28

```
1  __int64 __fastcall cipher_crypt(unsigned __int8 *a1, const unsigned __int8 *a2, int a3, unsigned __int8 *a4)
2  {
3    __int64 result; // rax
4    unsigned __int8 v5; // [rsp+3h] [rbp-Dh]
5    unsigned int i; // [rsp+4h] [rbp-Ch]
6    int v7; // [rsp+8h] [rbp-8h]
7    int v8; // [rsp+Ch] [rbp-4h]
8
9    LOBYTE(v8) = 0;
10   LOBYTE(v7) = 0;
11   for ( i = 0; ; ++i )
12   {
13     result = i;
14     if ( (int)i >= a3 )
15       break;
16     v8 = (unsigned __int8)(v8 + 1);
17     v7 = (unsigned __int8)(a1[v8] + v7 + 1);
18     v5 = a1[v8];
19     a1[v8] = a1[v7];
20     a1[v7] = v5;
21     a4[i] = a1[(unsigned __int8)(a1[v8] + a1[v7])] ^ a2[i];
22   }
23   return result;
24 }
```

Image 29

根据逆向结果编写解密脚本 `solve_rev_again.py`，用修改版 RC4 解密 `C_8_0`：

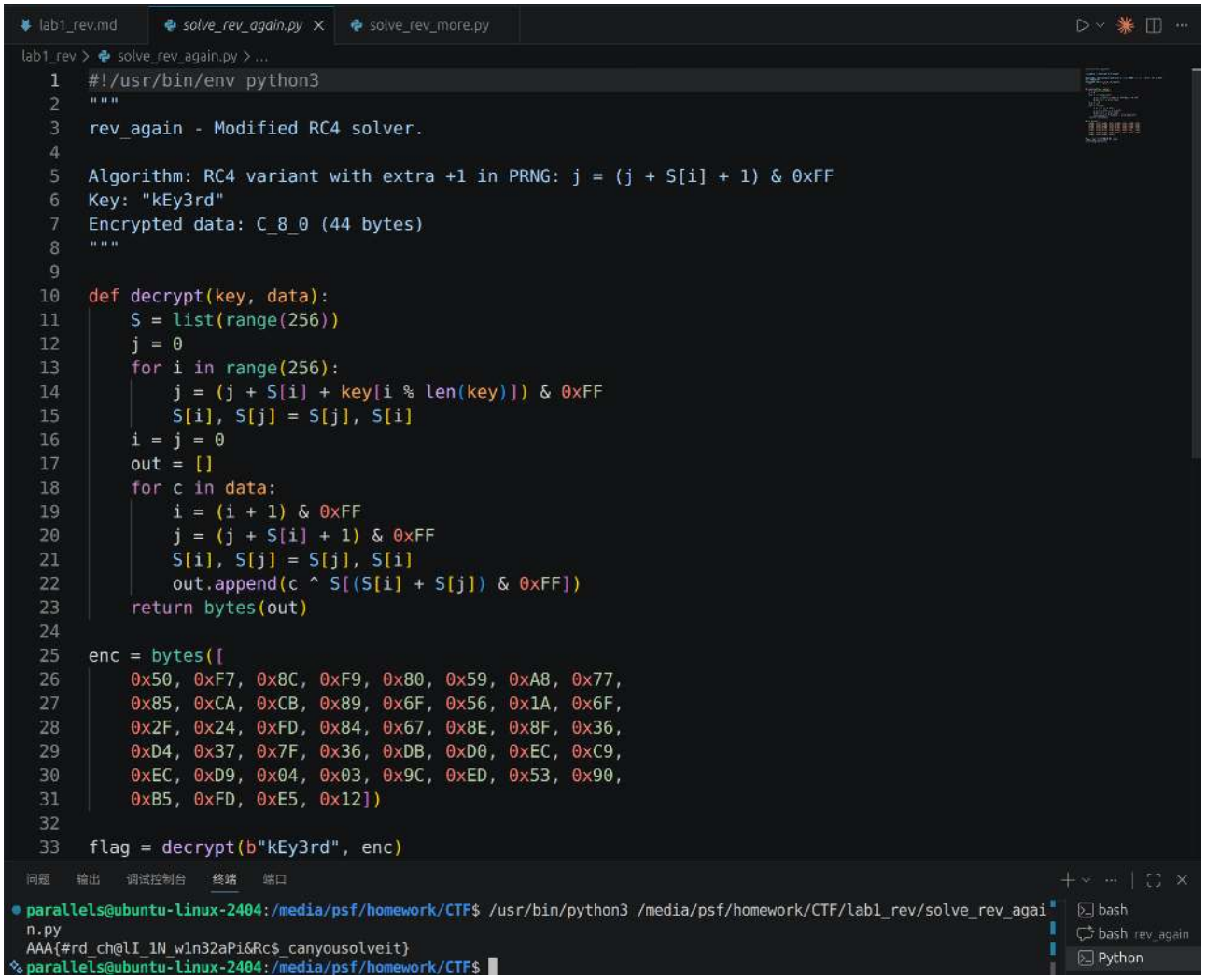

Image 30

Image 31