“Binary” Basics

逆向基础-1 @ f0rm2l1n

Outline

- 程序？可“执行”文件
- ELF 的编译、链接
- ELF 的装载、运行
- ELF 的交互、调试
- **ELF 的逆向**

程序？可执行文件

Raised Question

为什么计算机可以执行给定的程序呢？

因为任何程序都将最终转化为「指令」的形式由计算机执行

1+1 C Example

```c
int a = 1;
int b = 1;
int c = a + b;
```

高级语言
source code

```
mov     DWORD PTR [rbp-0xc],0x1
mov     DWORD PTR [rbp-0x8],0x1
mov     edx,DWORD PTR [rbp-0xc]
mov     eax,DWORD PTR [rbp-0x8]
add     eax,edx
mov     DWORD PTR [rbp-0x4],eax
```

汇编语言
assembly code

```
c745f401000000
c745f801000000
8b55f4
8b45f8
01d0
8945fc
```

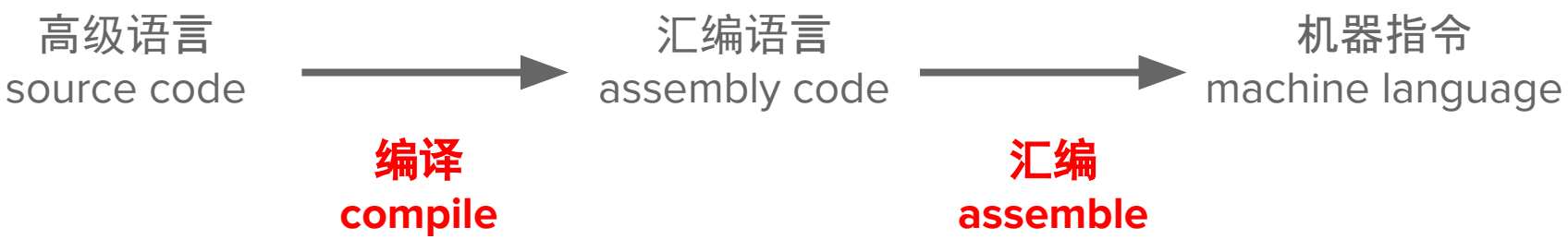
指令
instructions

1+1 C Example

```
int a = 1;
int b = 1;
int c = a + b;
```

```
mov     DWORD PTR [rbp-0xc],0x1
mov     DWORD PTR [rbp-0x8],0x1
mov     edx,DWORD PTR [rbp-0xc]
mov     eax,DWORD PTR [rbp-0x8]
add     eax,edx
mov     DWORD PTR [rbp-0x4],eax
```

```
c745f401000000
c745f801000000
8b55f4
8b45f8
01d0
8945fc
```



1+1 C Example with ARM

```c
int a = 1;
int b = 1;
int c = a + b;
```

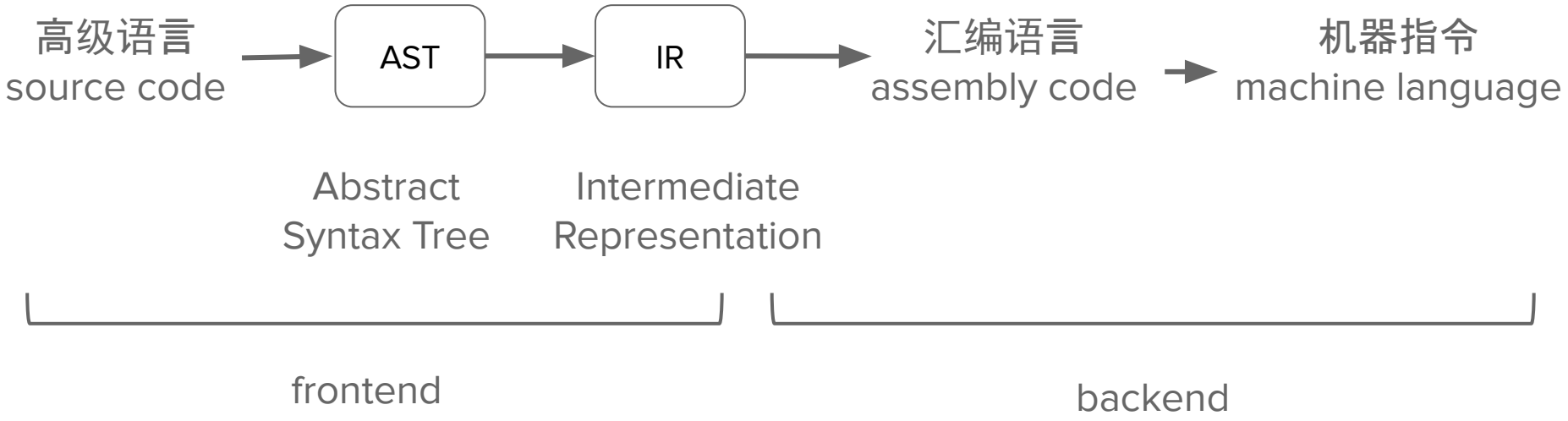
```
mov     DWORD PTR [rbp-0xc],0x1
mov     DWORD PTR [rbp-0x8],0x1
mov     edx,DWORD PTR [rbp-0xc]
mov     eax,DWORD PTR [rbp-0x8]
add     eax,edx
mov     DWORD PTR [rbp-0x4],eax
```

```
c745f401000000
c745f801000000
8b55f4
8b45f8
01d0
8945fc
```

```
mov     w0, #0x1
str     w0, [sp, #20]
mov     w0, #0x1
str     w0, [sp, #24]
ldr     w1, [sp, #20]
ldr     w0, [sp, #24]
add     w0, w1, w0
str     w0, [sp, #28]
```

```
52800020
b90017e0
52800020
b9001be0
b94017e1
b9401be0
0b000020
b9001fe0
```

Compile frontend & backend



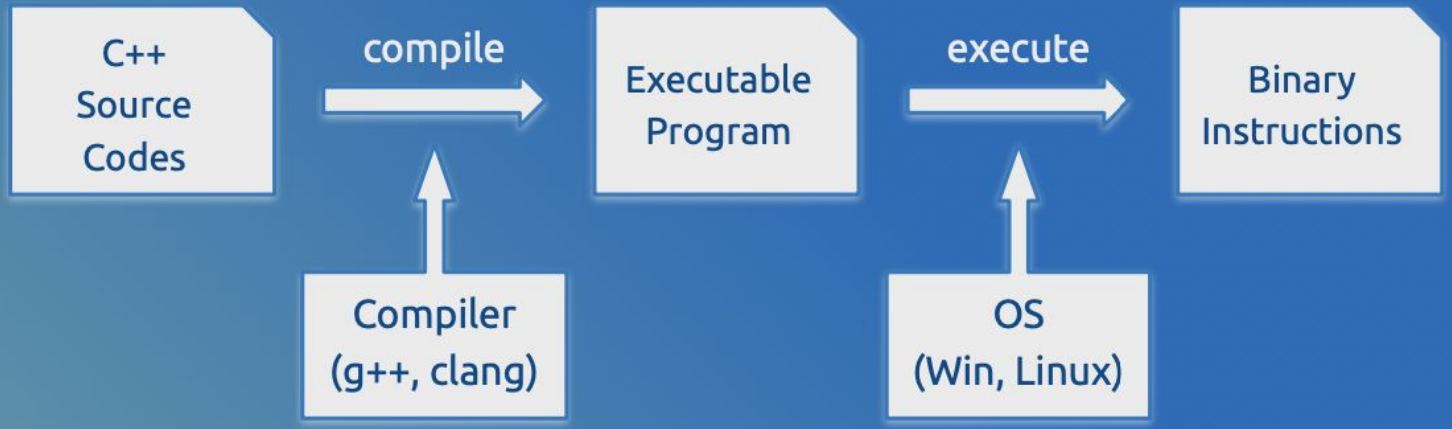
Compiled V.S. Interpreted

- **编译执行**
  上述通过编译器 (compiler) 将代码转化为机器指令格式的程序，进而执行

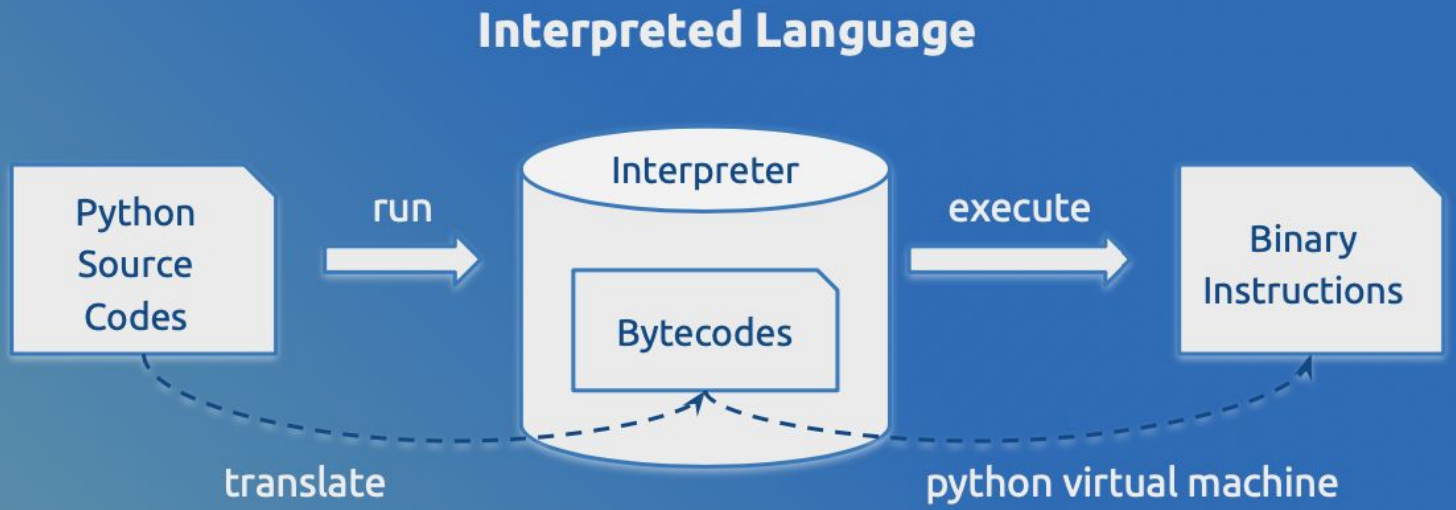
- **解释执行**
  通过解释器 (interpreter) 将代码转化为 VM 格式的程序（如字节码），进而在 VM 上执行

Compiled vs Interpreted

Compiled Language



Compiled vs Interpreted



碎碎念 ——“用户态可执行程序”

https://gitee.com/openharmony

用户态可执行文件

PE/PE32+
(Port Executable)

Mach-O
(Mach Object)

ELF
(Executable and Linkable Format)

https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats

Revisit ELF in Lab0

ELF (Executable and Linkable Format) is a common standard file format for “executables” in **Linux (or Unix-like)** systems

- 通过命令行工具静态检视 ELF 文件
  - `file`
  - `objdump`
  - `readelf`

ELF 的编译、链接

ELF的编译、链接 (C为例)

- 编译(汇编):从源代码到目标文件
- 链接:从目标文件到可执行文件

ELF的编译、链接 (C为例 cont,)

```c
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

```
gcc/clang hello.c
```

ELF的编译 (预处理)

```
gcc/clang -E hello.c -o hello.c.i
```

https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html

“essentially text substitution”

- 头文件包含
- 宏展开与替换

ELF的编译 (编译)

```
gcc/clang -S hello.c -o hello.s
```

- -S: Compile only; do not assemble or link
- -Ox: 优化等级
- -g: 启用调试
- ……

ELF的编译 (编译前端)

- 生成 clang AST

```
clang -Xclang -ast-dump -S hello.c
```

- 生成 LLVM IR

```
clang -Xclang -emit-llvm -S hello.c -o hello.ll
```

- *生成 gcc IR（以及其他细节）*

```
gcc -fdump-tree-all-graph -S hello.c
```

ELF的编译 (编译后端)

- 从 LLVM IR 到汇编代码

```
llc hello.ll -o hello.s
```

- 从汇编代码到目标文件 (object file)

```
llvm-mc -filetype=obj hello.s -o hello.o
```

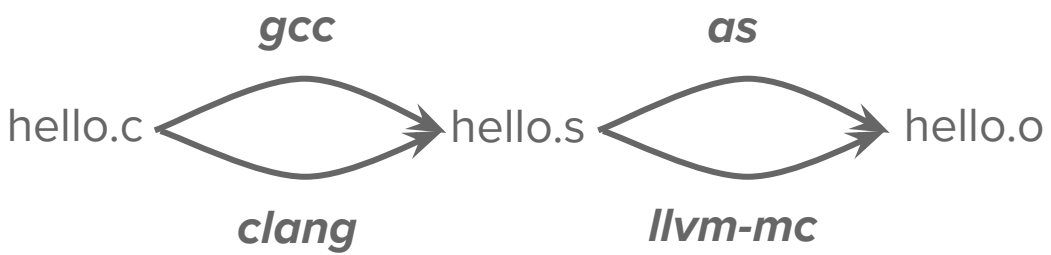
- 一步到位

```
llc -filetype=obj hello.ll -o hello.o
```

ELF的编译 (编译后端 cont.)

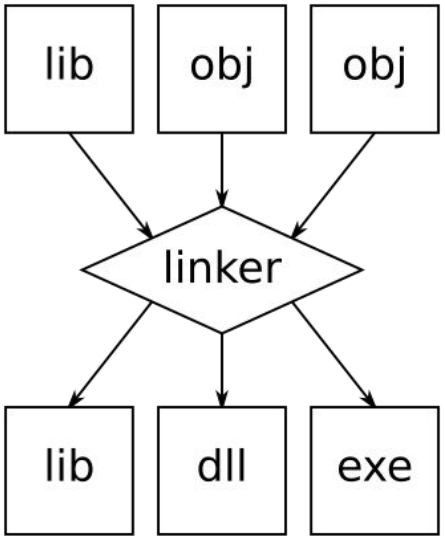
- 从汇编代码到目标文件 (object file) GCC 版本

```
as hello.s -o hello.o
```

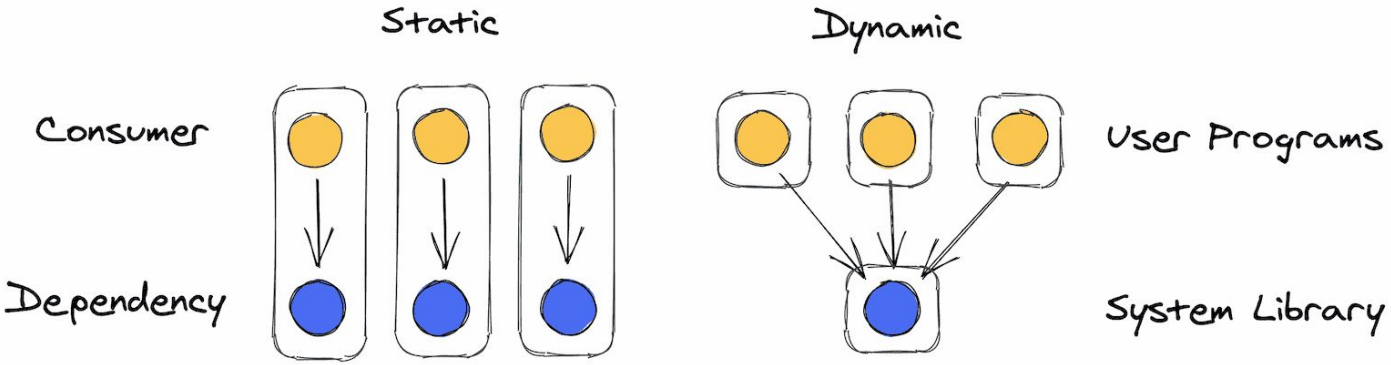


ELF的链接

- 目标文件 hello.o 的段
- 目标文件 hello.o 足以包含程序运行的信息了么？
- 符号解析与重定位
  - “essentially merging”
- LTO: Link-Time-Optimization



静态链接与动态链接



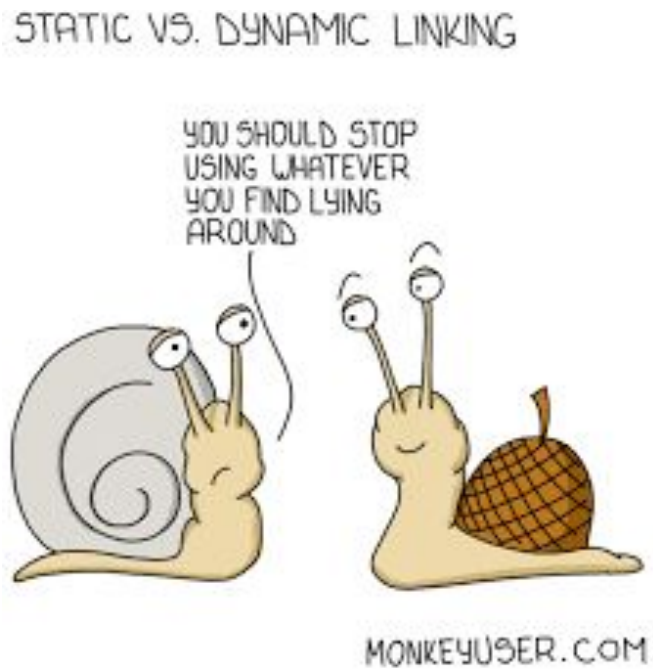
https://website-git-main.dumanjacob.vercel.app/writing/static-and-dynamic-linking

静态链接与动态链接 example

- 可执行文件大小比较
- 调用 *printf* 代码比较
- “运行时的内存布局比较”
- 优点、缺点

动态链接的PLT与GOT

- **PLT:** Procedure Linkage Table
- **GOT:** Global Offset Table
- lazy binding optimization
  以及 full-relro 保护



GNU LIBC 和 LD

- ldd 指令
- 通过指定 loader 来执行程序
- libc 及其版本
  - 糟糕的不向前兼容

ELF 程序的装载和运行

程序到进程

- 可执行程序 (Program) 是静态，文件的概念
- 进程 (Process) 是动态、系统运行时的概念
  - 进程和线程 (Thread)
- execve 系统调用

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

ELF程序的生命周期

Proposed Questions

- C/C++程序对应进程的起点是? 是 main 么?

- C/C++程序对应进程的终点是? main 结束后进程就结束了么?

before & after example

```c
#include <stdio.h>
#include <stdlib.h>

__attribute__((constructor)) void func1() {
    printf("Before main\n");
}

__attribute__((destructor)) void func2() {
    printf("After main\n");
}

int main(int argc, char *argv[])
{
    printf("During main\n");
}
```

attribute黑魔法 https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Function-Attributes.html

ELF程序的生命周期 (起点)

- 静态链接程序：
  内核以可执行文件 e_entry 位置 (即_start) 作为起点
  https://elixir.bootlin.com/linux/latest/source/fs/binfmt_elf.c#L1225

- 动态链接程序：
  内核以 interpreter 文件的 e_entry 位置作为起点
  https://elixir.bootlin.com/linux/latest/source/fs/binfmt_elf.c#L1200
  loader 负责解析和装载其他符号，进而再跳转到给定可执行文件的 _start

- 等了解交互/调试之后进行 revisit

_start

https://codebrowser.dev/glibc/glibc/sysdeps/x86_64/start.S.html#_start

- glibc 代码 (汇编构筑)
- 携带 *main* 符号跳转 *__libc_start_main* 函数

__libc_start_main

https://codebrowser.dev/glibc/glibc/csu/libc-start.c.html#234

- 完成各类和目标 ELF 有关的初始化
- 内联 *__libc_start_call_main*
- 最终跳往 main 符号
- main 结束后调用 exit

基地址与 ASLR

- PIE 动态链接可执行程序基地址随机保护
  https://elixir.bootlin.com/linux/latest/source/fs/binfmt_elf.c#L1089
- 无论静态/动态 (是否 PIE)，栈地址随即化保护
  https://elixir.bootlin.com/linux/latest/source/arch/x86/kernel/process.c#L997
- 通过 /proc/sys/kernel/randomize_va_space 控制随机化

ELF 程序的交互、调试

通过命令行人工与程序交互

- 绝对路径 / 相对路径
  - -h / --help
  - manual
  - PATH 路径

- 通过虚拟机或者沙箱进行交互
  - https://firejail.wordpress.com/
  - https://github.com/google/nsjail

通过编程与程序交互

https://gitee.com/zjuicsr/ssec24summer-stu/wikis

- 重定向构建特殊字符作为输入
- C 管道编程
- python subprocess 库
- *python pwntools 库*

strace & ltrace

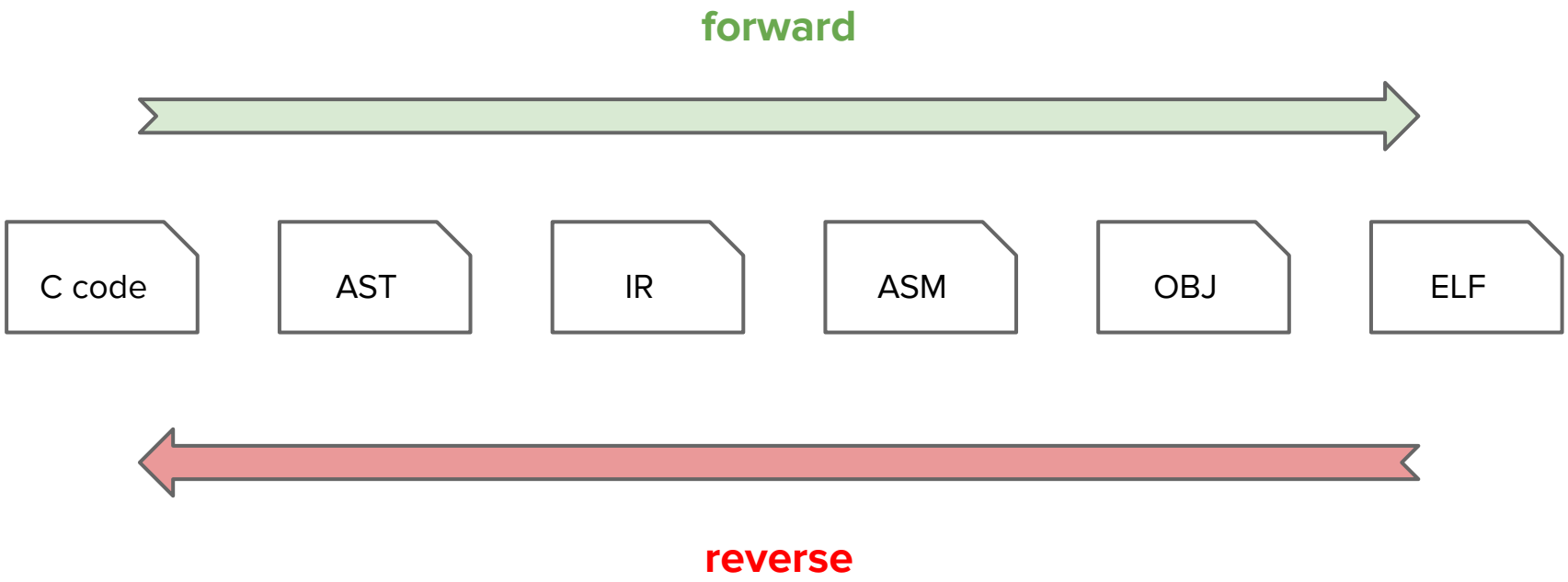
- system calls / library calls tracer
- 基于 **ptrace** 实现
- Examples

GDB: GNU DeBugger

- 调试模式
  - 调试器执行模式
  - attach 模式
  - *remote 模式*
- 常用调试功能
  - 执行断点
  - 硬件断点
  - 查看寄存器 / 内存
  - set 修改寄存器 / 内存
- gdb 插件

ELF 程序的逆向

ELF 的正向和逆向



Reverse Engineering

- Basically, RE (Reverse Engineering) is about **interacting with given objects** (e.g. ELF executables) and try to figure out what they are doing.
- “RE is **30%** guess work, **70%** hard work.”
- 个人观点：逆向工程（及其竞赛）题目是各类题目中相对“友好”的

Why RE?

- **Bad Aspect - 破解版、外挂**

- **Good Aspect - 未知攻、焉知防**
  - 恶意样本分析
    https://cloud.tencent.com/developer/news/501963
  - 软件保护技术

ELF 的静态逆向: 反汇编器与反编译器

- **反汇编**: 机器指令 => 汇编指令（查表、准确）
  - objdump

- **反编译**: 汇编指令 => 编程语言（分析/特征匹配/启发式 ... 、往往不准确）
  - IDA Pro (https://hex-rays.com/ida-pro/)
  - Binary Ninja (https://binary.ninja)
    - ***with free version***
  - Ghidra (https://github.com/NationalSecurityAgency/ghidra)
  - Cutter / radare (https://github.com/rizinorg/cutter)
  - 大语言模型 ;D https://mlm.lingyiwanwu.com/

ELF 的静态逆向技巧

对于静态链接目标

- 特征 + 猜测尽可能恢复库函数符号

... 静态去符号是纯恶心人 ...

对于符号恢复的静态 or 动态链接目标

- 关注特定**常量**和**字符串**
- 关注**输入**和**输出函数**
- 关注分支、比较指令
- 关注可能涉及加密解密的特殊运算（位运算、异或、取余）

ELF 的动态逆向

「**可运行**」和「**可调试**」是高效解决逆向问题的必备

- 许多逆向赛题都需要「纯静态」的方式解决，程序可能依赖特定的架构/设备
- “if it can run, it can be cracked”
- 通过运行时的结果解决静态逆向时的疑惑

ELF 逆向 example

- 参见随堂材料中的 crackme-ext

- 静态做法 / 动态做法

Takeaways

- 编译执行程序的正向和逆向
- ELF 程序的运行和调试
- 基础的静态+动态 ELF 逆向