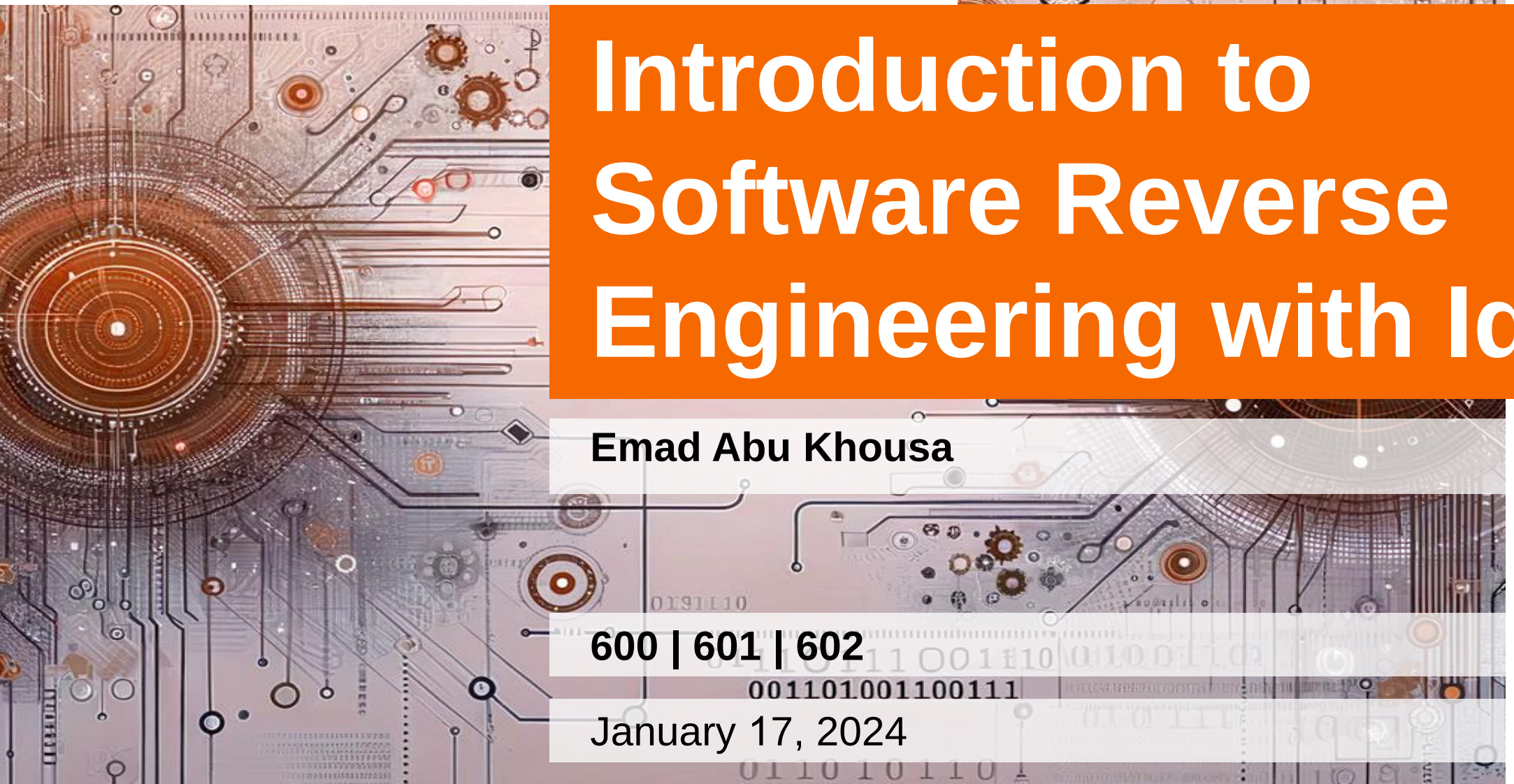
RIT

جامعة روتشستر الأمريكية للتكنولوجيا في دبي

A Global American University in Dubai



Introduction to Software Reverse Engineering with Ida

Emad Abu Khousa

600 | 601 | 602

00110100110011 1

January 17, 2024

Agenda

- **Introduction to Software Reverse Engineering (SRE)**
  - Introduce reverse engineering, its significance in cybersecurity, and basic C program compilation.
- **Exploring IDA**
  - Demonstrate IDA installation, usage, and assembly code analysis
- **More C to Assembly**
  - Explore function calls and loop structures in assembly using IDA Pro.
- **Reverse Engineering: From Binary to C**
  - Disassemble a complex binary, reconstructing its C code representation.
- **Summary and Q&A**

Introduction to Software Reverse Engineering (SRE)

The Reverse Engineering Process

- **Defining Reverse Engineering:**
  - Understanding a system by analyzing its binary form.
- **From Source Code to Binary:**
  - Example: Display the "Hello RIT Dubai!" C code.
  - Compilation: How the C code becomes machine code.
- **The Challenge:**
  - Presented with only the binary, how do we understand the program?
- **Objective:**
  - Task: Translate binary back into human-readable code to comprehend its functionality.

Check your understanding:

1. **What is the primary goal of binary reverse engineering?**
   a) To create new software binaries from scratch.
   b) To decode software binaries back to their original source code and logic.
   c) To encrypt software binaries for security purposes.
   d) To improve the graphical interface of software programs.

2. **What role does IDA play in software reverse engineering?**
   a) It is used to write new software programs.
   b) It is an interactive disassembler that helps analyze and understand binaries.
   c) It encrypts software binaries.
   d) It converts high-level languages into machine code.

Check your understanding:

3. In the context of reverse engineering, what is the significance of understanding the transition from C to assembly language?

a) It helps in improving the speed of the C program.

b) It is crucial for understanding how high-level language constructs are represented in machine code.

c) It is only important for developing new programming languages.

d) It is irrelevant as long as the C program runs correctly.

4) What is a key challenge in reverse engineering a complex binary using IDA?

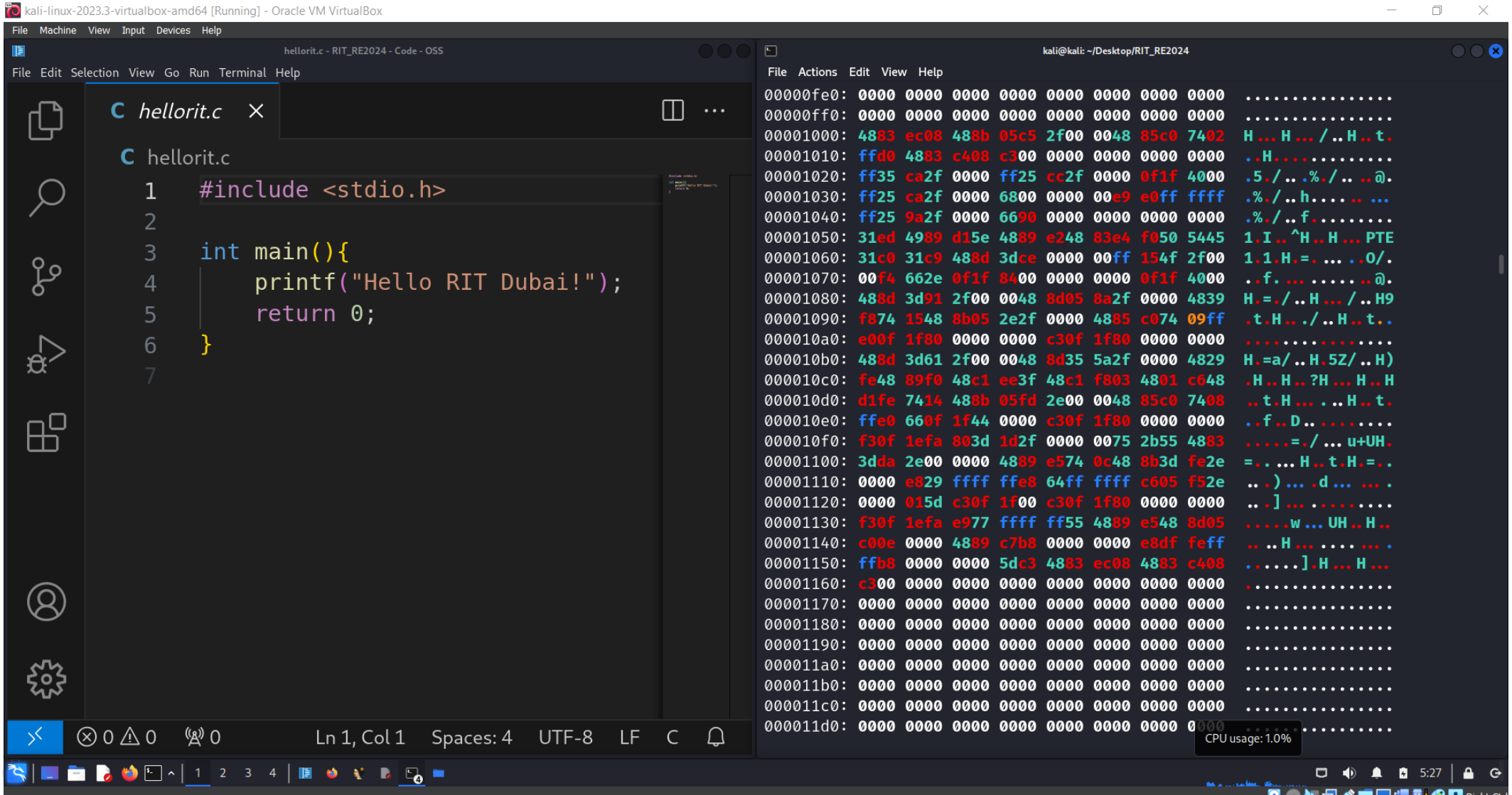
a) Disassembling the binary and interpreting the assembly language.

b) Identifying proprietary algorithms used in the binary.

c) Enhancing the performance of the binary.

d) Translating the binary directly into a scripting language like Python.

The Reverse Engineering Process



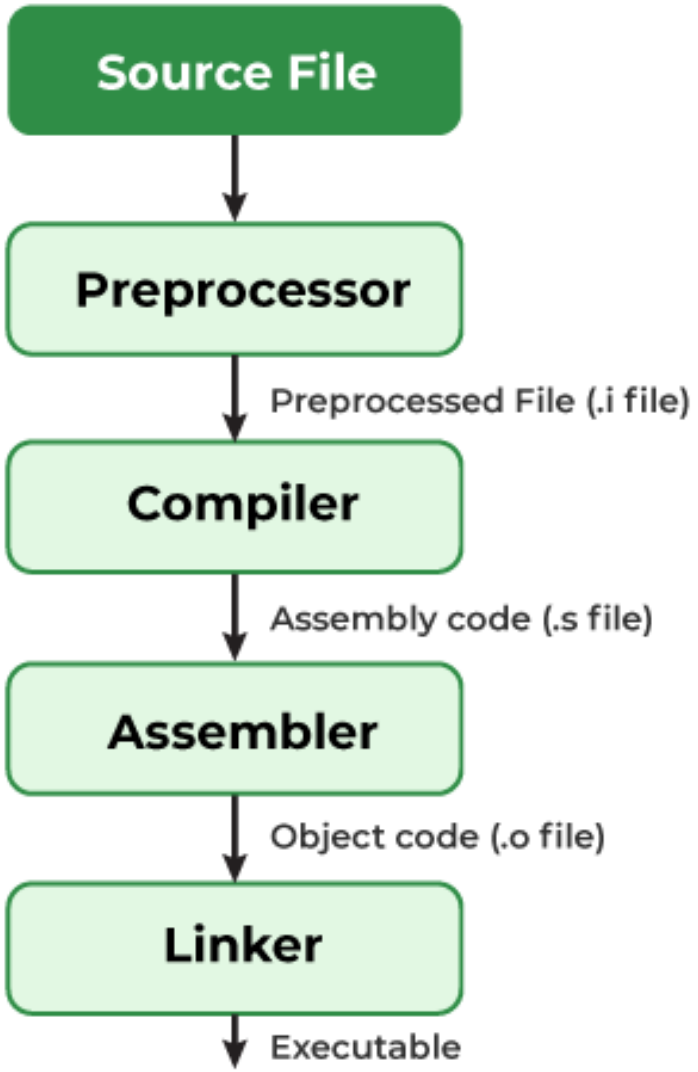
Hands-On

- 1. Download the Source Code
  - Access the source code from GitHub:
  - URL: https://github.com/RITDubaiCSEC202/CSEC202-Spring2024
  - File to Download: hellorit.c
- 2. Compilation to Executable
  - For Windows: Compile to a PE (Portable Executable) file.
  - For Linux: Compile to an ELF64 (Executable and Linkable Format) file.

$gcc filename.c -o filename

Hands-On

- **3. Detailed Compilation Process**
  - Understanding the four phases of C program compilation:
  - Pre-processing
  - Compilation
  - Assembly
  - Linking
  - Compile with detailed output for each phase.

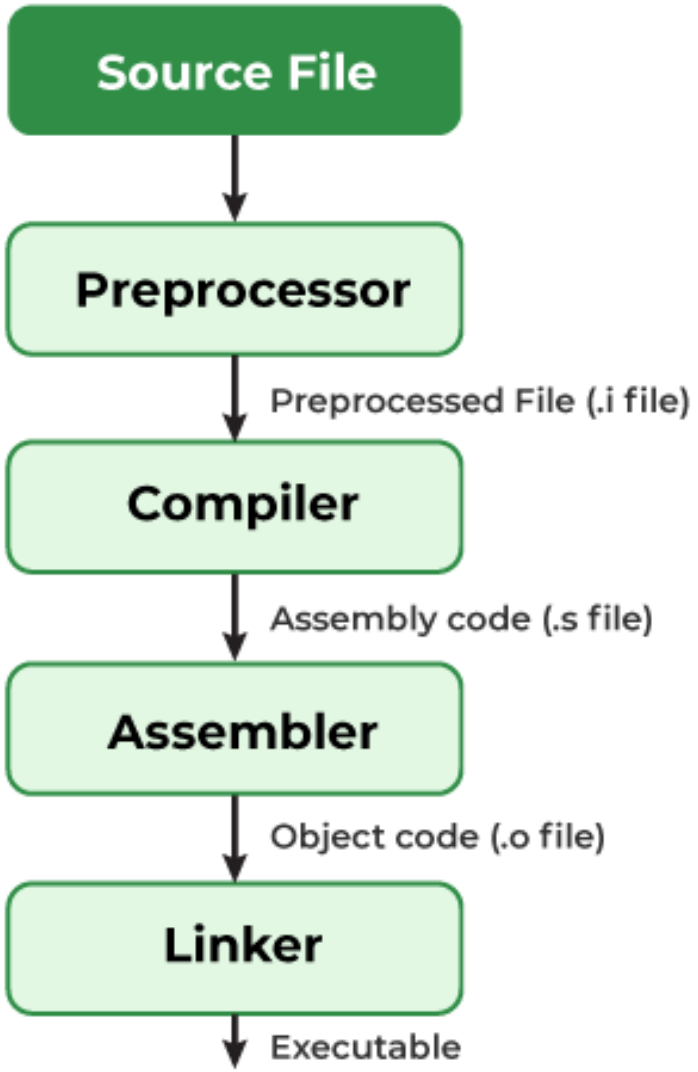


$gcc -Wall -save-temps filename.c -o filename

Hands-On

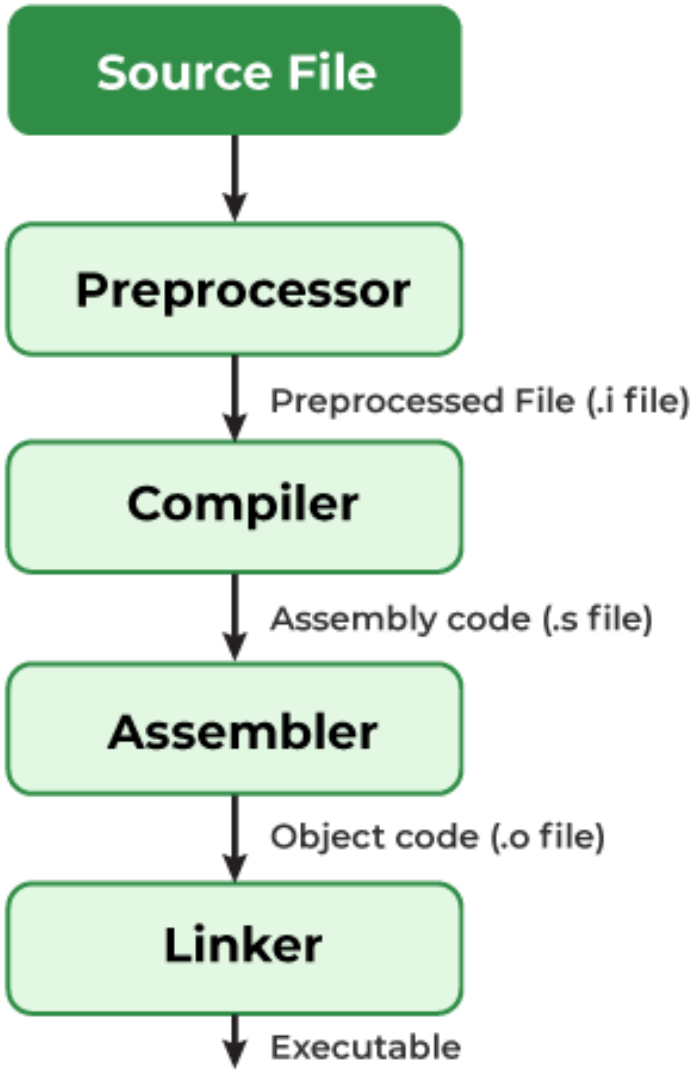
- **4. Display the source code of each file**

$gcc -Wall -save-temps filename.c -o filename



Hands-On

- **5. Go Intel**
  - Compile with GCC and Generate Assembly Code: Use the -S option with GCC to generate assembly code. To specify Intel syntax, use the -masm=intel option. For example:
- **$ gcc -S -masm=intel your_program.c -o your_program.asm**
- This command compiles your_program.c and generates an assembly file your_program.asm in Intel syntax.



Why We Reverse Engineer?

- **For Cybersecurity Pros:**
  - Triage by Hand | Uncover Intent | Craft Defenses
  - Cybersecurity Forensics: Examine malware's activities, data theft, and command and control (C2) pathways.
- **Diverse Applications:**
  - Understand Applications | Firmware Analysis | Shared Object Exploration
- **It's Just Fun!:**
  - The Puzzle: Enjoy the challenge of decoding the unknown..

Check your understanding:

What is the primary goal of binary reverse engineering in cybersecurity, and why is it important?

a) To enhance the graphical interface of binary files for better user experience.

b) To translate binary files into high-level languages like C for a deeper understanding of their functionalities, crucial for analyzing malware and building defense strategies.

c) To increase the processing speed of binary files by converting them into more efficient code.

d) To create backups of binary files for data recovery purposes.

Scenario-Based Question:

"Imagine you've discovered an unknown application behaving suspiciously on your network. What steps would you take to understand its functionality and potential impact on your systems?"

a) Run an antivirus scan.
b) b) Update the application.
c) c) Analyze the application's binary code to understand its operations.
d) d) Ignore it as a false alarm.

Scenario-Based Question:

"A cybersecurity analyst receives a piece of software with no documentation. The software is critical for a legacy system. What approach should the analyst take to ensure the software can be safely integrated without compromising system security?"

a) Test the software on different systems.
b) Disassemble the software to understand its inner workings and potential vulnerabilities.
c) Consult with the original software developers.
d) Immediately integrate the software into the system.

Scenario-Based Question:

"In what situation would understanding assembly language be crucial for a software engineer?“

a) When designing a user-friendly interface.
b) When optimizing the performance of a high-level application.
c) When trying to understand the low-level operations of a compiled program.
d) When writing a report on software development trends.

Scenario-Based Question:

"You are tasked with improving the security of an existing software application but have no access to its source code. What would be your initial step?“

a) Redesign the application from scratch.
b) Use reverse engineering to understand the current software's structure and vulnerabilities.
c) Implement additional firewalls and security protocols.
d) Conduct user training on cybersecurity best practices.

Scenario-Based Question:

"If a cybersecurity team needs to analyze an advanced persistent threat (APT) that has infiltrated their network, which skill is most likely to be pivotal in understanding the threat's mechanism?”

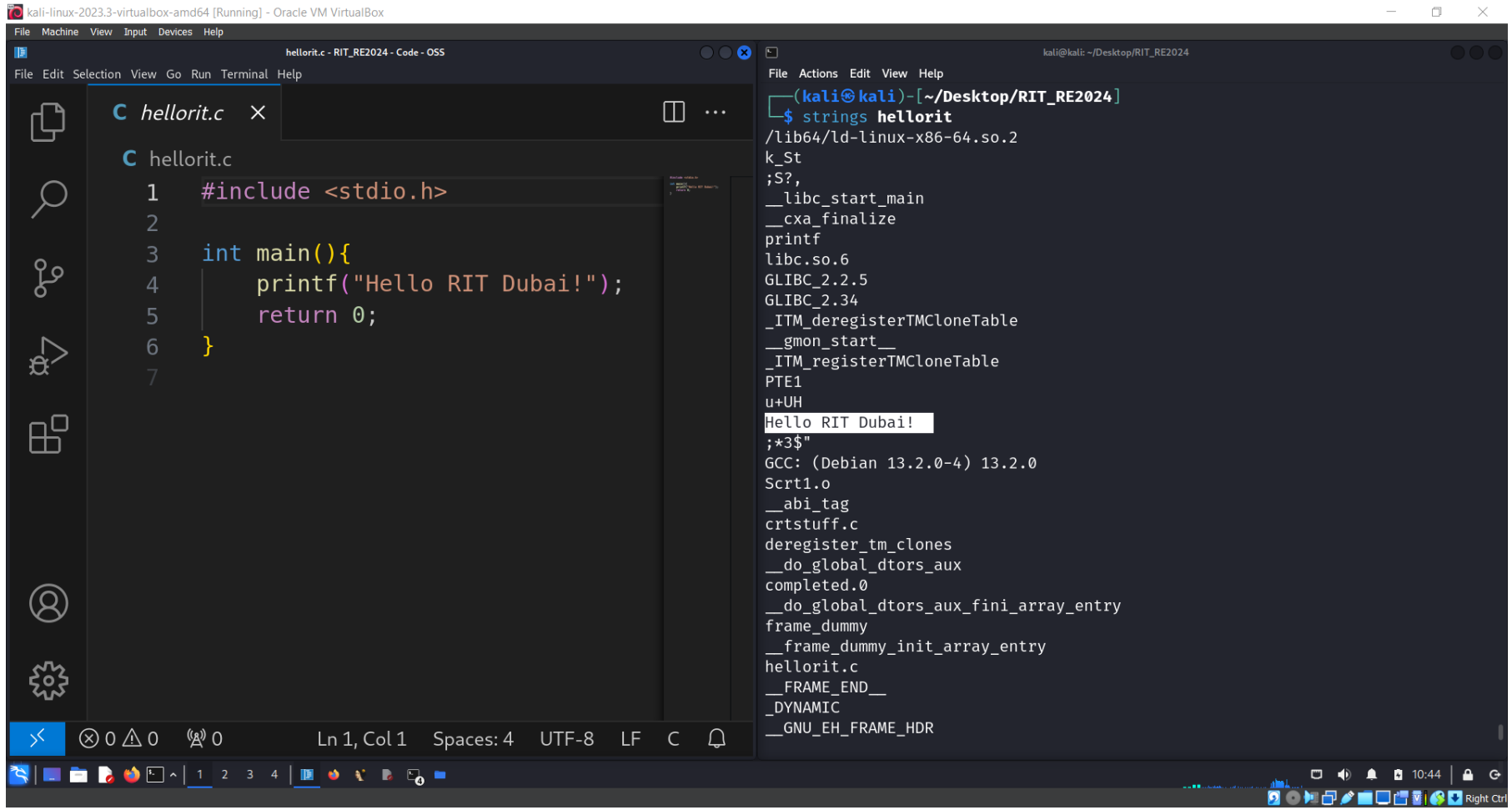
a) Ability to design new network architectures.
b) Proficiency in reverse engineering to dissect the malware.
c) Expertise in hardware repair.
d) Skills in website design.

Scenario-Based Question:

"You are tasked with improving the security of an existing software application but have no access to its source code. What would be your initial step?“

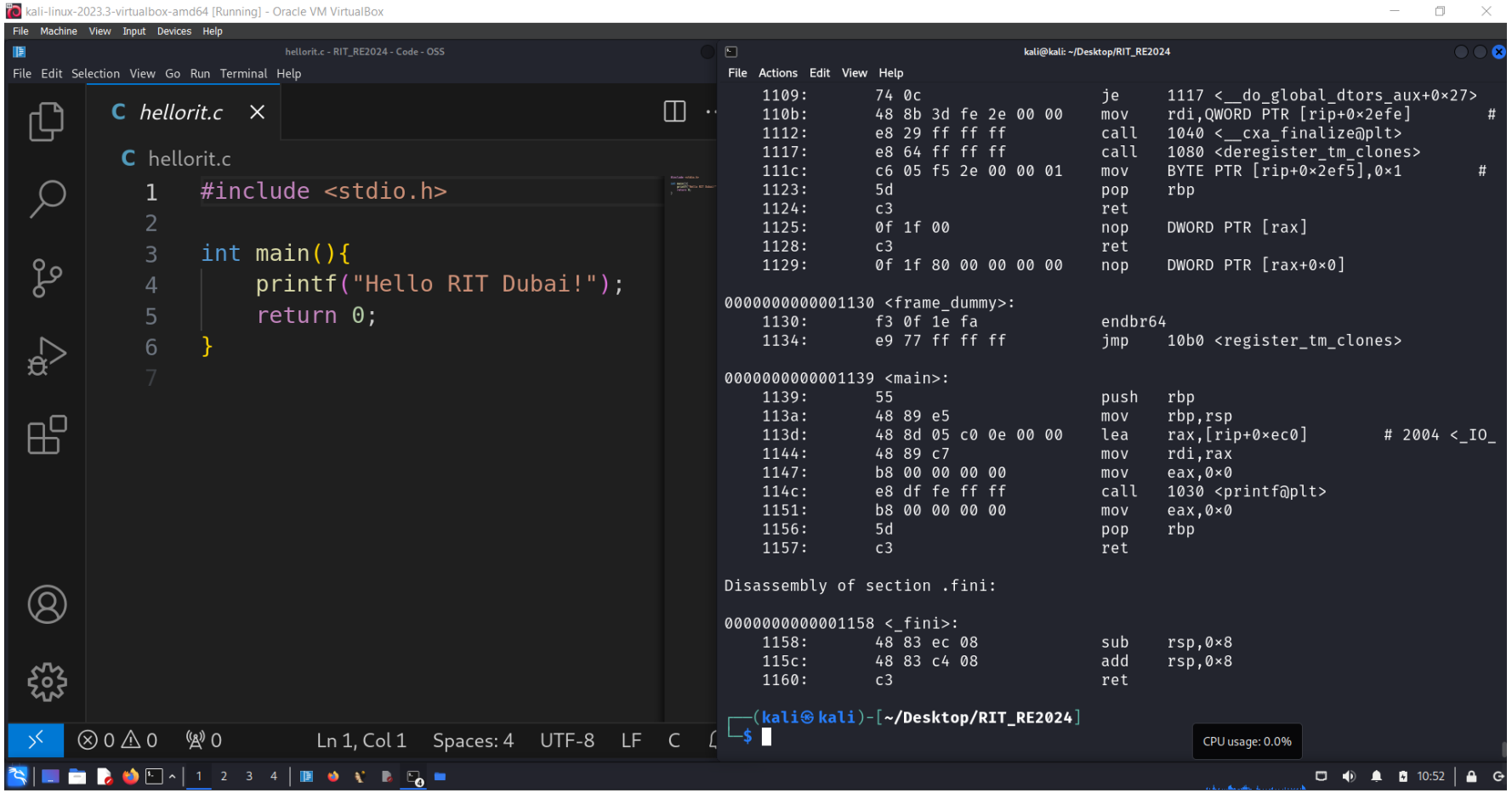
a) Redesign the application from scratch.
b) Use reverse engineering to understand the current software's structure and vulnerabilities.
c) Implement additional firewalls and security protocols.
d) Conduct user training on cybersecurity best practices.

Reverse Engineering 101: Binary Analysis (1/2)



$ strings hellorit

Reverse Engineering 101: Binary Analysis (2/2)



$ objdump -d -Mintel hellorit | less

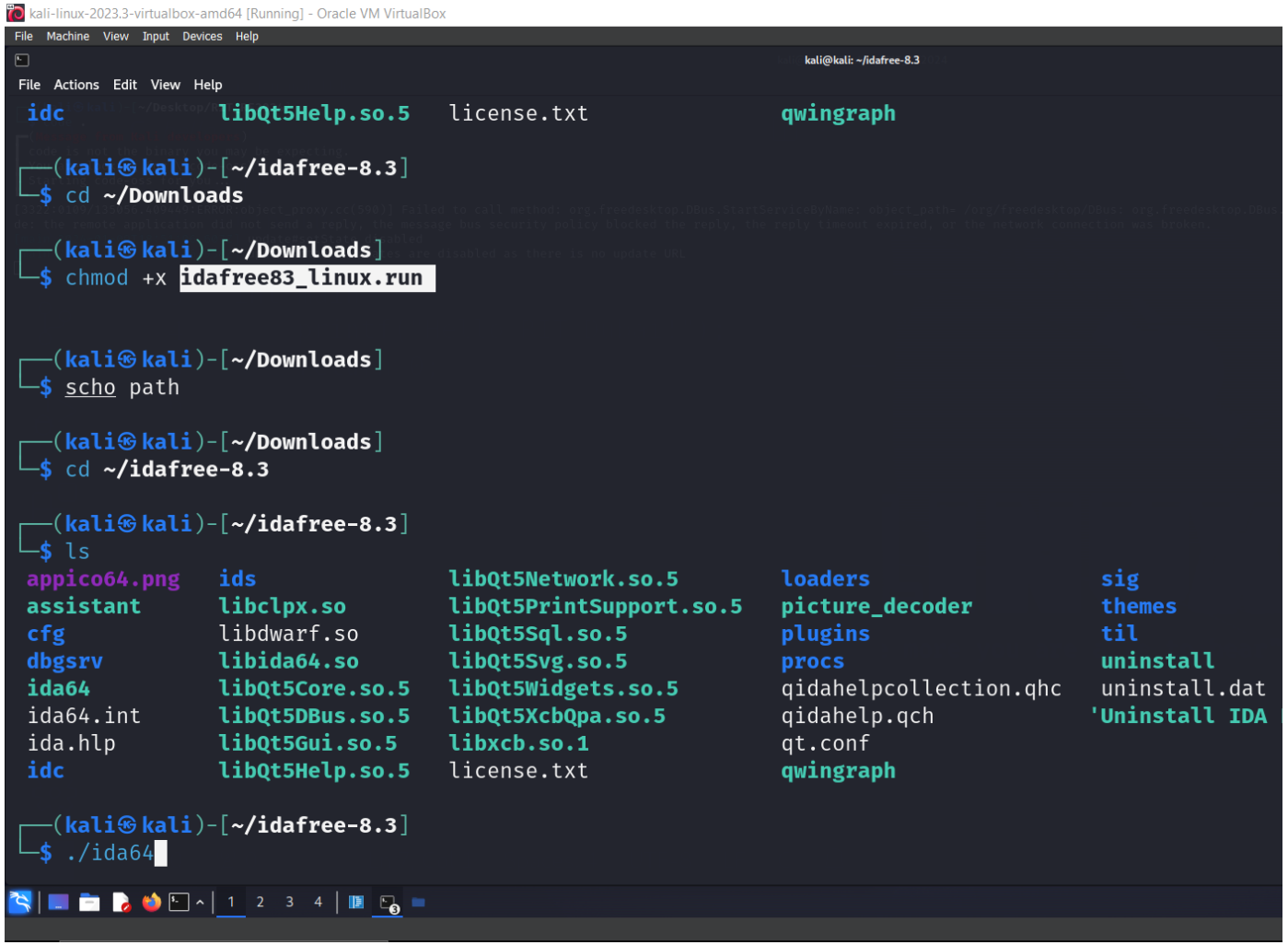
Exploring IDA

Advanced reverse engineering disassemblers

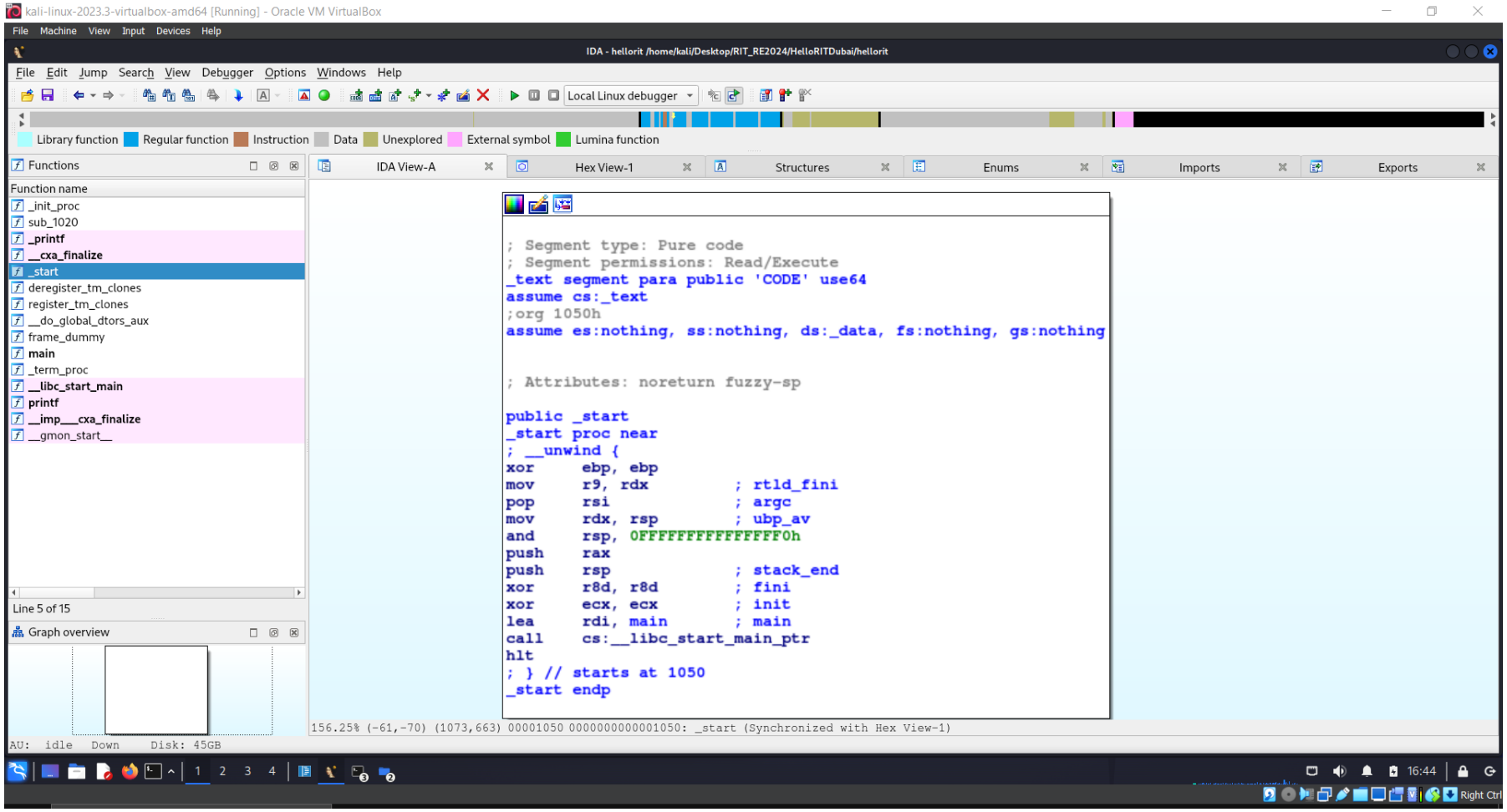
Installing and running IDAFree

- Download IDAfree from https://hex-rays.com/ida-free/

- $ chmod +x idafree83_linux.run
- $ ./idafree83_linux.run
- $ cd ~/idafree-8.3
- $ ./ida64



Meet IDA


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
IDA - hellorit /home/kali/Desktop/RIT_RE2024/HelloRITDubai/hellorit
Local Linux debugger
Library function
Regular function
Instruction
Data
Unexplored
External symbol
Lumina function
Functions
Function name
_init_proc
sub_1020
_printf
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
main
_term_proc
__libc_start_main
printf
__imp___cxa_finalize
__gmon_start__
Line 5 of 15
Graph overview
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 1050h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
; Attributes: noreturn fuzzy-sp
public _start
_start proc near
; __unwind {
xor ebp, ebp
mov r9, rdx ; rtld_fini
pop rsi ; argc
mov rdx, rsp ; ubp_av
and rsp, 0FFFFFFFFFFFFFFF0h
push rax
push rsp ; stack_end
xor r8d, r8d ; fini
xor ecx, ecx ; init
lea rdi, main ; main
call cs:__libc_start_main_ptr
hlt
; } // starts at 1050
_start endp
156.25% (-61,-70) (1073,663) 00001050 0000000000001050: _start (Synchronized with Hex View-1)
AU: idle Down Disk: 45GB
16:44
Right Ctrl

Start Function

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 1050h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: noreturn fuzzy-sp

public _start
_start proc near
; __unwind {
xor     ebp, ebp
mov     r9, rdx             ; rtld_fini
pop     rsi                 ; argc
mov     rdx, rsp            ; ubp_av
and     rsp, 0FFFFFFFFFFFFFFF0h
push    rax
push    rsp                 ; stack_end
xor     r8d, r8d            ; fini
xor     ecx, ecx            ; init
lea     rdi, main           ; main
call    cs:__libc_start_main_ptr
hlt
; } // starts at 1050
_start endp
```

Application Binary Interface (ABI)

Foo (1, 2)

Argument 1 = 1 ➔ saved in **rdi**

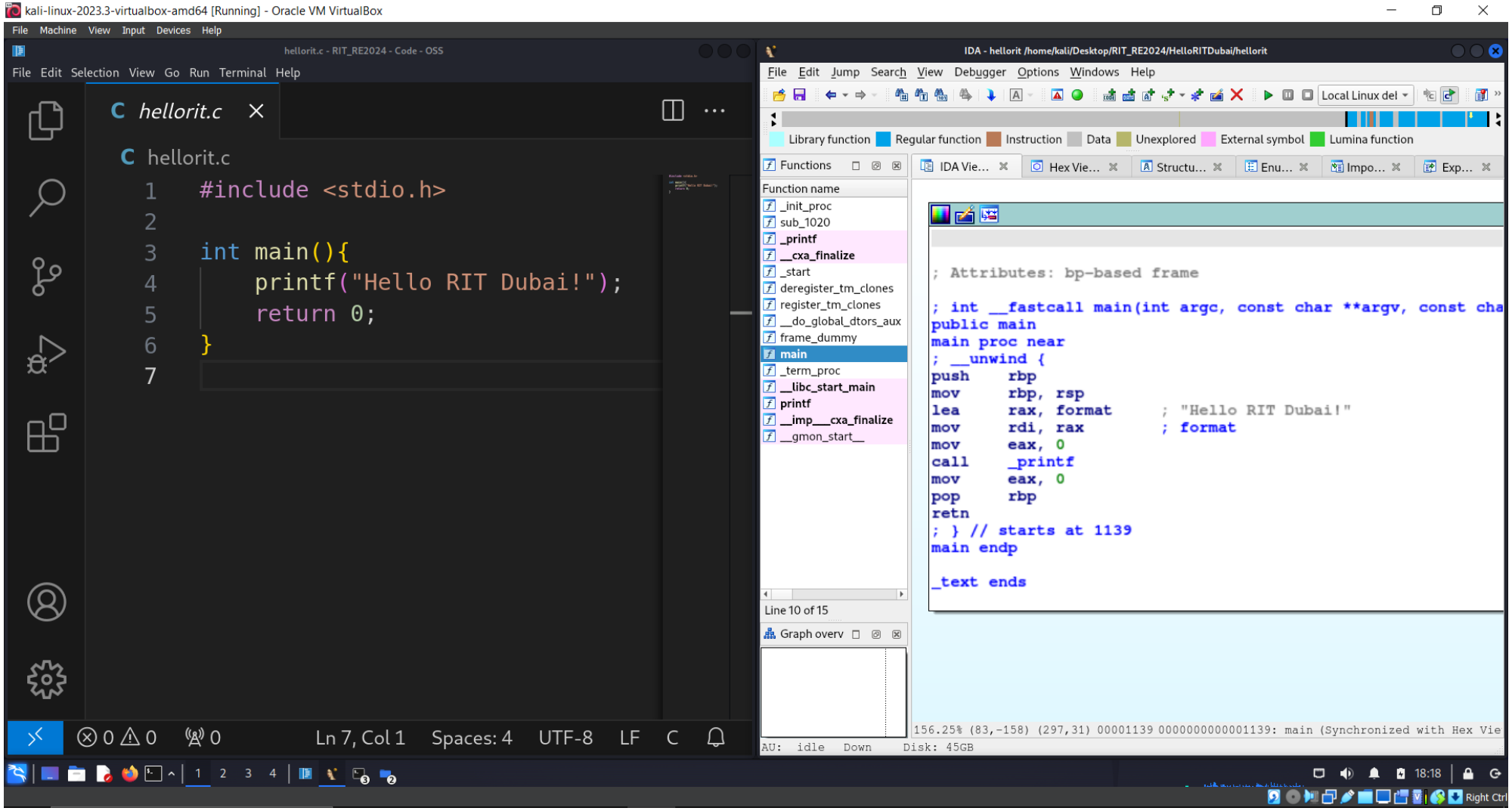
Argument 2 = 2 ➔ saved in **rsi**

- rdi: used to pass 1st argument to functions
- rsi: used to pass 2nd argument to functions
- rdx: used to pass 3rd argument to functions
- rcx: used to pass 4th argument to functions

Start Function

```
.text:0000000000001050 _start          proc near               ; DATA XREF:
LOAD:0000000000000018↑o
.text:0000000000001050 ; __unwind {
.text:0000000000001050                 xor     ebp, ebp
.text:0000000000001052                 mov     r9, rdx         ; rtld_fini
.text:0000000000001055                 pop     rsi             ; argc
.text:0000000000001056                 mov     rdx, rsp        ; ubp_av
.text:0000000000001059                 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:000000000000105D                 push    rax
.text:000000000000105E                 push    rsp             ; stack_end
.text:000000000000105F                 xor     r8d, r8d        ; fini
.text:0000000000001062                 xor     ecx, ecx        ; init
.text:0000000000001064                 lea     rdi, main       ; main
.text:000000000000106B                 call    cs:__libc_start_main_ptr
.text:0000000000001071                 hlt
.text:0000000000001071 ; } // starts at 1050
.text:0000000000001071 _start          endp
```

Main Function



Main Function

```
.text:0000000000001139 ; int __fastcall main(int argc, const char **argv, const char **envp)
.text:0000000000001139                 public main
.text:0000000000001139 main            proc near               ; DATA XREF: _start+14↑o
.text:0000000000001139 ; __unwind {
.text:0000000000001139                 push    rbp
.text:000000000000113A                 mov     rbp, rsp
.text:000000000000113D                 lea     rax, format     ; "Hello RIT Dubai!"
.text:0000000000001144                 mov     rdi, rax        ; format
.text:0000000000001147                 mov     eax, 0
.text:000000000000114C                 call    _printf
.text:0000000000001151                 mov     eax, 0
.text:0000000000001156                 pop     rbp
.text:0000000000001157                 retn
.text:0000000000001157 ; } // starts at 1139
```

Main Function

```
main
push    rbp
mov     rbp, rsp
lea     rax, format
mov     rdi, rax
mov     eax, 0
call    _printf
mov     eax, 0
pop     rbp
retn
```

Prologue or Stack Frame Setup:

- The "**push rbp**" instruction saves the base pointer onto the stack, preserving the current stack frame.
- With "**mov rbp, rsp**", we set the base pointer for the new stack frame to the current stack pointer's position.

Main Function

main

```
push    rbp
mov     rbp, rsp
lea     rax, format
mov     rdi, rax
mov     eax, 0
call    _printf
mov     eax, 0
pop     rbp
retn
```

Preparing the Message:

- The "**lea rax, format**" instruction loads the effective address of the string "Hello RIT Dubai!" into the "rax" register. Essentially, it's preparing the data for usage.

- "**mov rdi, rax**" then transfers the address of our message into the "rdi" register, which is used as the first argument to the "printf" function.

Main Function

```
main
push    rbp
mov     rbp, rsp
lea     rax, format
mov     rdi, rax
mov     eax, 0
call    _printf
mov     eax, 0
pop     rbp
retn
```

Function Call:

- "**mov eax, 0**" clears the "eax" register, which is a convention before a function call to indicate the number of vector registers used.
- The "**call _printf**" instruction is the function call to "printf", which will output our message to the console.

Main Function

main

```
push    rbp
mov     rbp, rsp
lea     rax, format
mov     rdi, rax
mov     eax, 0
call    _printf
mov     eax, 0
pop     rbp
retn
```

Clean Up and Return:

- After the function call, "**mov eax, 0**" sets the return value of "main" to 0, indicating successful execution.

Epilogue:

- "**pop rbp**" restores the base pointer from the stack, cleaning up the stack frame.
- Finally, "**retn**" returns control to the caller, effectively ending the function..

Summary and Q&A