Software reverse engineering summary

Reverse engineering is the process of extracting important design factors and information regarding engineering from anything man made.

Software reverse engineering is the system analysis process to identify a system’s components and its interrelationships and create representations of the system in another form or at higher levels of abstraction.

Why do Reverse engineering?

To learn about a program’s structure and logic

The two categories of reverse engineering

* Security related reversing
* Software development related reversing

Uses of reverse engineering

* To cope with complexity
* To recover lost information
* To detect side effects
* To facilitate reuse

Re-engineering

It is the process of analysis and change whereby a system is modified by first reverse engineering and then forward engineering.

Activities in re-engineering are

* Automatic restructuring
* Automatic and semi-automatic transformation
* Design recovery and reimplementation
* Code reverse engineering and forward engineering
* Data reverse engineering and schema migration
* Migration of legacy systems to modern platforms

Goals of reengineering

* Port to another platform
* Design extraction
* Exploitation of new technology

Reengineering Techniques

* Restructuring: automatic conversion from unstructured to structured code
* Data reengineering: integrating and centralizing multiple databases
* Refactoring: A refactoring is a software transformation that preserves the external behavior of the software and improves the internal structure of the software

Reasons for refactoring

* To improve the software design
* To counter code decay and increase its lifetime
* To reduce costs of software maintenance
* To find bugs and write more robust code
* To prepare for/ facilitate future customization

Types of Re-factoring

* Code restructuring
* Data restructuring

Problems with re-structuring are:

* + Loss of comments
  + Loss of documentation
  + Heavy computational demands

NOTE 2a

Reversers must literally be aware of anything that comes between the program **source code** and the **CPU**.

Why do reversers need knowledge of low-level languages?

That is because the low-level aspects of a program are often the only thing you have to work with as a reverser.

A key concept about reversing is that reversing tools such as disassemblers or de-compilers never actually provide the answers—they merely present the information.

**low-level software**

It encompasses development tools such as compilers, Interpreter, linkers, and debuggers, infrastructure software such as operating systems, and

low-level programming languages such as assembly language.

It is the layer that isolates software developers and application programs from the physical hardware.

Assembly language is the lowest level in the software chain.

Machine code and assembly language are two different representations of the same thing.

A CPU reads machine code, which is nothing but sequences of bits that contain a list of instructions for the CPU to perform.

Assembly language is simply a textual representation of those bits.

Each assembly language command is represented by a number, called the operation code, or opcode.

Object code is essentially a sequence of opcodes and other numbers used in connection with the opcodes to perform operations.

CPUs constantly read object code from memory, decode it, and act based on the instructions embedded in it.

they use an **assembler** program to translate the textual assembly language code into binary code.

In the other direction and more relevant to our narrative, a **disassembler** does the exact opposite.

It reads object code and generates the textual mapping of each instruction in it.

**Disassemblers** are a key tool for reversers.

A compiler is a program that takes a source file and generates a corresponding machine code file.

Compilers of traditional (non-bytecode-based) programming languages such as C and C++ directly generate machine-readable object code from the textual source code.

The biggest hurdle in deciphering compiler-generated code is the optimizations applied by most modern compilers.

Compilers for high-level languages such as Java generate a bytecode instead of an object code.

Bytecodes are similar to object codes, except that they are usually decoded by a program, instead of a CPU.

Virtual machine contains interpreter that is used to interpret the code to the intended CPU.

An interpreter is a type of computer program that directly executes instructions written in a programming or scripting language.

**Benefits of using bytecode**

platform independence: The virtual machine can be ported to different platforms This enables running the same binary program on any CPU as long as it has a compatible virtual machine.

This means that theoretically software developers do not need to worry about platform compatibility

An operating system is a program that manages the computer, including the hardware and software applications.

Note 2b

Code level techniques provide detailed information on a selected code chunk.

Reversing sessions can be divided into two separate phases:

* System-Level Reversing: This is really a kind of large-scale observation of the earlier program. This techniques help determine the general structure of the program and sometimes even locate areas of interest within it.

It involves running various tools on the program and utilizing various OS services to obtain information, inspect program executables, and track program input and output.

* Code-Level Reversing: Code level approaches offer comprehensive details on a particular code segment.

Extracting this information requires a mastery of reversing techniques along with a solid understanding of software development, the CPU, and the operating system

Code-level reversing observes the code from a very low-level, seeing every little detail of how the software operates

**Reversing Tools**

Some of the basic categories of tools that are used in reverse engineering are:

* System-monitoring tools: System-monitoring tools can monitor networking activity, file accesses, registry access, and so on.
* Dis-assemblers: Disassemblers are programs that take a program’s executable binary as input and generate textual files that contain the assembly language code for the entire program or parts of it.
* De-buggers: A debugger is a software tool that lets programmers see their code in action while it's executing.
* De-compilers: A de-compiler takes an executable binary file and attempts to produce readable high-level language code from it.

Note 3

Program structure is the thing that makes software, an inherently large and complex thing, manageable by humans.

Developing a robust and reliable product rests primarily on two factors:

* each component box is well implemented and reliably performs its duties, and
* each box has a well-defined interface for communicating with the outside world

In most reversing scenarios, the first step is to

* determine the component structure of the application and
* the exact responsibilities of each component.

The largest building block for a program is the module.

There are two basic types of modules that can be combined together to make a program:

* + static libraries
  + and dynamic libraries.

**Static libraries** make up a group of source-code files that are built together and represent a certain component of a program.

**Dynamic libraries** (called Dynamic Link Libraries, or DLLs in Windows) are similar to static libraries, except that they are not embedded into the program.

There are two basic code-level constructs that are considered the most fundamental building blocks for a program; procedures and objects.

* + A **procedure** is a piece of code, usually with a well-defined purpose, that can be invoked by other areas in the program. It used in procedure-based program.
  + **Object**. This where a system is divided into objects. This process is called object-oriented design (OOD), and is considered by many to be the most popular and effective approach to software design currently available.

Data Management: A program deals with data.

It requires two perspectives:

* + the high-level perspective as viewed by software developers and
  + the low-level perspective that is viewed by reversers.

High-level data constructs such as variables and the most common types of data structures.

Variables. This is the key to managing and storing data by software developer.

User-Defined Data Structures. User-defined data structures are simple constructs that represent a group of data fields, each with its own type.

Control flow statements are statements that affect the flow of the program based on certain values and conditions.

Here is a brief overview of the basic high-level control flow constructs:

* + **Conditional blocks**: Conditional code blocks are implemented in most programming languages using the if statement. They allow for specifying one or more condition that controls whether a block of code is executed or not.
  + **Switch blocks**: Switch blocks (also known as n-way conditionals) usually take an input value and define multiple code blocks that can get executed for different input values.
  + **Loops**: Loops allow programs to repeatedly execute the same code block any number of times. A loop typically manages a counter that determines the number of iterations already performed or the number of iterations that remain.

High-level languages were made to allow programmers to create software without having to worry about the:

* + specific hardware platform on which their program would run and
  + all kinds of annoying low-level details that just aren’t relevant for most programmers.

The challenge is to sift through this information with enough understanding of the high-level language used and to try to reach a close approximation of what was in the original source code.

C programming. C is a compiled language, meaning that to run the program you must run the source code through a compiler that generates platform-specific program binaries.

C code is relatively easy to reverse because it is fairly similar to the machine code.

The most common low-level data management constructs are

* registers,
* stacks, and
* heaps

they relate to higher-level concepts such as variables and parameters.

Registers are small chunks of internal memory that reside within the processor and can be accessed very easily, typically with no performance penalty whatsoever.

Unfortunately, managing registers and loading and storing data from RAM to registers and back certainly adds a bit of complexity to assembly language code.

These are the types of complexities added by the use of registers.

While reversing, it is important to try and detect the nature of the values loaded into each register.

Note 4

Compilers and interpreters take human-readable code and convert it to computer-readable machine code.

An Interpreted Language is a type of programming language where the code is compiled on the fly each time the program is run.

Interpreters run through a program line by line and execute each command.

Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

Compiled languages are converted directly into machine code that the processor can execute.

Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Reversibility Of C++

* The C++ programming language is an extension of C, and shares C’s basic syntax.
* C++ introduce support for object-oriented programming.
* The core feature introduced in C++ is the class.
* Reversing code written in C++ is very similar to working with C
* except that emphasis must be placed on deciphering the program’s class hierarchy and on properly identifying class method calls

Reversibility Of Java

* Java is an object-oriented, high-level language that is different from other languages such as C and C++
* It is not compiled into any native processor’s assembly language, rather into the Java bytecode representation.
* the process of reversing Java classes is usually much simpler than with native code.

Reversibility Of C#

* C# was developed by Microsoft as a Java-like object-oriented language that aims to overcome many of the problems inherent in C++.
* C# was introduced as part of Microsoft’s .NET development platform.
* C# programs are compiled into an intermediate bytecode format (similar to the Java bytecode) called the Microsoft Intermediate Language (MSIL).
* MSIL programs run on top of the common language runtime (CLR), which is essentially the .NET virtual machine.

Reversibility Of Python

* Python is an interpreted high-level general-purpose programming language.
* It is a multi-paradigm programming language that support OOP and structured programming.
* Many of its features support functional programming and aspect-oriented programming.
* Similar to C++ and Java, Python code is compiled to a bytecode and then interpreted to machine code.
* Python program can be reversible using appropriate tools.
* However, recently obfuscation is a popular technique used by programmers to protect their code.
* De-compilers are used to reverse engineer Python program depending on the type of obfuscation applied.
* For some type of obfuscation, the de-compiler is unable to understand the input provided to it and then the game is over.

Advantages of compiled languages

* Programs that are compiled into native machine code tend to be faster than interpreted code.

Disadvantages of compiled languages

The most notable disadvantages are:

* + Additional time needed to complete the entire compilation step before testing
  + Platform dependency of the generated binary code.

Advantages of interpreted languages

* Interpreted languages tend to be more flexible
* often offer features like dynamic typing and smaller program size.
* Also, because the code is designed to be executed by interpreters, the code itself is platform independent.

Disadvantages of interpreted languages

The most notable disadvantage is typical execution speed compared to compiled languages.

Note 5

To look at even the most basic assembly language code, you must become familiar with IA-32 registers

For most purposes, the IA-32 has eight generic registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP

IA-32 processors have a special register called EFLAGS that contains all kinds of status and system flags.

The status flags, are used by the processor for recording its current logical state, and are updated by many logical and integer instructions in order to record the outcome of their actions.

Instructions usually consist of an opcode (operation code), and one or two operands.

Operands represent data that is handled by the specific instruction

A compiler is a program that takes one representation of a program as its input and produces a different representation of the same program.

In most cases, the input representation is a text file containing code that complies with the specifications of a certain high-level programming language.

The output representation is usually a lower-level translation of the same program.

The average compiler consists of three basic components.

The front end is responsible for deciphering the original program text and for ensuring that its syntax is correct and in accordance with the language’s specifications.

The optimizer improves the program in one way or another, while preserving its original meaning.

Finally, the back end is responsible for generating the platform- specific binary from the optimized code emitted by the optimizer.

A listing file is a compiler-generated text file that contains the assembly language code produced by the compiler.

Most compilers support the generation of listing files during the compilation process.

For some compilers, such as GCC, this is a standard part of the compilation process

In such compilers, requesting a listing file simply means that the compiler must not delete it after the assembler is done with it.

In other compilers (such as the Microsoft or Intel compilers), a listing file is an optional feature that must be enabled through the command line.

An execution environment is the component that actually runs programs.

Execution environments are especially important to reversers because their architectures often affect how the program is generated and compiled

There are two basic types of execution environments, which are;

* Virtual machines (software execution environment) and
* Microprocessors (hardware execution environment).

Three common virtual machine architectures are the Java Virtual Machine (JVM) that runs Java programs,

the Common Language Runtime (CLR) that runs Microsoft .NET applications and

The Python Virtual Machine (VM) that executes Python bytecode, which is generated from Python source code.

Programs that run on virtual machines have several significant benefits compared to native programs executed directly on the underlying hardware:

* Platform isolation: Because the program reaches the end user in a generic representation that is not machine-specific, it can theoretically be executed on any computer platform for which a compatible execution environment exists.
* Enhanced functionality: When a program is running under a virtual machine, it can (and usually does) benefit from a wide range of enhanced features that are rarely found on real silicon processors.

Just-in-time compilation is an alternative approach for running bytecode programs without the performance penalty associated with interpreters. The idea is to take snippets of program bytecode at runtime and compile them into the native processor’s machine language before running them.

These snippets are then executed natively on the host’s CPU.

A microprocessor was a collection of digital circuits that could perform a variety of operations and was controlled using machine code that was read from memory.

The processor’s runtime consisted simply of an endlessly repeating sequence of

* reading an instruction from memory,
* decoding it, and
* triggering the correct circuit to perform the operation specified in the machine code.

The general architecture of modern IA-32 processors is as follows:

* IA-32 Compatible Processors
* Intel NetBurst
* μops (Micro-Ops)
* Pipelines
* Branch Prediction