267. SOFTWARE REVERSE ENGINEERING AS A POWERFUL TOOL FOR CODE ANALYSIS

Nikitin K.K.

*Belarusian State University of Informatics and Radioelectronics
Minsk, Republic of Belarus*

Kaspiarovich N.G. – Senior Lecturer

The paper investigates the concept of reverse engineering, including its application domains, classifications, as well as the approaches and technologies commonly used. The analysis of a simple program is presented.

Software reverse engineering is the process of analyzing a software system to understand its architecture, underlying implementation, when original source code is not available. The process is complex because after the program is compiled, information about the high-level abstractions used in programming is lost. Reverse engineering is applied in the following cases: system integration to ensure compatibility, information security to study malware, documentation reconstruction, lost source code recovery, bug fixing and support for unavailable programs with unavailable source code.

For further delving into the technologies and approaches used, the knowledge of the stages of compilation is essential. This process will be examined in greater detail below. The compilation process consists of the following stages [1]:

- Lexical analysis. The sequence of characters in the source file is converted into a sequence of language lexemes. At this stage, comments and formatting are removed.
- Syntax analysis. The sequence of lexemes is transformed into a parse tree according to the language rules. This tree represents the syntactic structure of the program.
- Semantic analysis. The parse tree is processed to verify that the program complies with the language rules, for example, mapping identifiers to their declarations and types, determining expression types, etc.
- Intermediate representation. Most compilers transform code into one or more Intermediate Representations which are used for optimization: expression simplification, removal of unused code, loop unrolling, inlining, etc. Often, during compilation, it is possible to specify a flag indicating the optimization level.
- Code generation. Considering all the hardware architectural features, object code is generated from the Intermediate Representation. It should be noted that this step is often preceded by assembly generation. At this stage, local variables are replaced by the use of the stack and registers. Next, for reverse engineering, it is needed to identify the variables and their data types based on the register and stack usage patterns.
- Linking. The final step in the process of producing an executable file consists of linking together all the project object files. Object file contains the following sections: .text – machinery code; .bss – uninitialized global variables, .data – initialized global variables; .rodata – constants and string literals, symbol table and relocation table [2].

Therefore, the program structure changes significantly. Regarding identifiers, the names of imported and exported functions, often global variables and sometimes other variables are retained. Common reverse engineering practices and tools are discussed below.

- Strings. As mentioned above, a binary file contains strings that were originally in the source code. Analyzing these strings can often be useful for understanding what the program eventually does.
- Static analysis. It is aimed at examining an executable file without running it. This is done with the help of disassembly – converting machine code into assembly instructions, a representation that remains difficult for humans to analyze. Therefore, some disassemblers have the ability to represent a program as a Control Flow Graph, in which code sections are grouped into interconnected blocks, making analysis simpler. In addition to disassembly, there is decompilation – the restoration of high-level code close to the original. To achieve this, decompilers analyze the Control Flow Graph and, as mentioned earlier, memory usage patterns to generate pseudocode, typically in C-like languages. It should be noted that this is the decompiler’s best guess at what the original C code might have looked like. However, due to Intermediate Representation optimizations, the generated pseudocode may differ significantly from the original. Also, due to the lack of identifier and type information, the decompiler is forced to restore variables and assign them new names that don't carry the original structural meaning, which complicates analysis. An example of a disassembler is objdump. Environments that combine a disassembler and decompiler include IDA Pro and Ghidra.
- Dynamic analysis. This method is aimed at examining an executable file while it's running, observing how it performs its functions. This is achieved through debugging, which allows stepping through program instructions and setting breakpoints to analyze the program at specific points during execution; for example, viewing the stack contents, which typically stores local variables. It is important to select breakpoints correctly. In external debuggers, this is accomplished through the memory address of the instruction to be stopped at; however, reverse engineering environments often have a built-in debugger, such as IDA Pro and Ghidra. External debuggers used in reverse engineering include GDB, Windbg, and x64dbg. In addition to debugging,

such approaches as analysis of API calls, system calls, and other interfaces, as well as the program's network activity, are also used.

- Symbolic execution [3]. Program analysis method aimed at identifying input data that triggers the execution of specific sections of code. For this purpose, symbols, similar to mathematical variables, are used instead of specific input data. By going through conditional branches of the program, the tools used for symbolic execution construct a complex logical expression. During execution, symbolic expressions are stored in processor registers. The resulting expression is passed to an SMT solver, such as Z3, which evaluates the resulting expression and determines the input data. Typically, when analyzing large programs, targeting specific branches is practiced. To achieve this, frameworks must be passed the address of the location in the program being searched for, which can be achieved using static analysis. It should be noted that for large programs, this approach is resource intensive. Examples of frameworks include KLEE and Angr.

To demonstrate the concepts behind decompilation and the impact of compiler optimizations on generated pseudocode, an example is provided below. A simple function for summing numbers in an array is written in C language and called in the *main* program (Figure 1). The program is then compiled using the GCC compiler with two different flags: O0 that stands for default, no optimization level and O2 that is standard high optimization level. Terminal commands are presented in Figure 2.

```
int sum(int* a, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

Figure 1 – C code for the considered function

```
gcc -O0 script.c -o script_O0
gcc -O2 script.c -o script_02
```

Figure 2 – GCC compilation command with O0 and O2 optimization flags

Then two generated PE files were analyzed using Ghidra software and two pseudo code functions shown on Figure 3 were obtained.

```
int sum(undefined8 param_1,undefined8 param_2,
int param_3,long param_4)
{
  undefined4 local_10;
  undefined4 local_c;
  local_c = 0;
  for (local_10 = 0; local_10 < param_3; local_10 = local_10 + 1) {
    local_c = local_c + *(int *)(param_4 + (long)local_10 * 4);
}
  return local_c;
}
```

```
int sum(undefined8 param_1
,undefined8 param_2,
int param_3,int *param_4)
{
  int *piVar1; int iVar2;
  if (0 < param_3) {
    iVar2 = 0;
    piVar1 = param_4 + param_3;
    do {
      iVar2 = iVar2 + *param_4;
      param_4 = param_4 + 1;
    } while (param_4 != piVar1);
    return iVar2;
  }
  return 0;
}
```

Figure 3 – Decompiled code of a function compiled with a flag O0 (on the left) and O2 (on the right)

In the first case, some correspondence is traced between the variables: sum variable *s* – *local_c*, loop iterator *i* – *local_10*. In the second case, instead of an integer loop iterator, *param_4* is used to store the address of the element, so the multiplication operations can be removed. The address of the last element – the condition for exiting the loop is calculated in *piVar* variable. As an optimization, code also has a condition for early exit from a function. What is important to note is that in the second pseudo code function the type of the second argument is determined correctly unlike the first function.

Therefore, this example confirms that decompilation is a guess, on how the source code might have looked like, which depends heavily on how this source code was compiled.

References:

1. Phases of a Compiler : [website]. – USA, 2026. – URL: https://www.geeksforgeeks.org/compiler-design/phases-of-a-compiler/ (date of access: 04.03.2026).

2. Reverse engineering tools: [website]. – USA, 2026. – URL: https://www.infosecinstitute.com/resources/reverse-engineering/reverse-engineering-tools/ (date of access: 04.03.2026).

3. Symbolic execution. The first sign of using angr : [website]. – USA, 2026. – URL: https://paranoid.security/blog/p/symbolic-execution-angr-05-08 (date of access: 07.03.2026).