Research and practice of software protection based on virtual shell technology

Long Zhang1*

1School of Information Engineering, Xi’an Eurasia University, Xi’an Shaanxi, 710065, China.
*Corresponding author’s e-mail: zhanglong@eurasia.edu

Abstract. Through software decompilation technology, decompiling software can obtain pseudocode of the program source code, causing more and more events of core technology leakage or threats to software intellectual property rights, especially, the proliferation of pirated software. Aiming at the problem of how to effectively protect the software, this article implements a virtual shell packer with the virtual shell technology based on the QT5 framework as the front-end page frame and the assembly language to write the interpreter. Based on this packer, the code of protected software would be decompiled into logically chaotic code in order to increase the difficulty of cracking. Experiments show that this technology can effectively prevent the original pseudo-code from being restored through decompilation. So the software products are effectively protected.

1. Introduction
As the state strengthens the protection of intellectual property rights, there are fewer and fewer blatant infringement cases. At the same time, it can be seen that there are still a large number of cracked software and pirated software on the Internet, but a small number of malicious technology analysts steal the core technology of software products through reverse analysis technology, which greatly damages the legitimate rights and interests of developers. This is extremely detrimental to the long-term development of the software industry, and the protection of software intellectual property rights still has a long way to go. Therefore, in addition to strengthening the protection of intellectual property rights from the legal level, the software industry is also trying to increase the difficulty of software cracking and decompilation through technical means. This article designs a set of custom coding, a packer and an interpreter, and add the custom-coded shell to the program to be protected by the shell method to prevent malicious decompilation during program execution. Then, convert the custom code into a normal program that the computer can recognize through the interpreter to realize the complete protection and running process of the program.

2. Relevant knowledge

2.1. Virtual shell protection technology
The essence of virtual shell protection technology is the conversion of instructions, from normal assembly instructions to customized instructions, thus hiding the original code. Compared with other shell protection technologies, this virtual shell protection method is more secure, because although
other shells hide functions, the executed code will not change, only static source code protection is 
protected in various ways. This virtual shell protection method is to replace the original instructions in 
the software, and realize the simulation execution of the original instructions through the custom 
instruction conversion. But now all interpreters on the market cannot interpret the custom instructions, 
so the virtual shell protection method needs to use their interpreter to restore the program code. 
Moreover, the content of the custom instruction set is also uncertain, it is much more difficult to 
analyse [1] the original instructions.

2.2. Software analysis method

2.2.1. Static analysis. Static analysis [2] is to load the program to be analyzed into the memory to 
simulate the loading program of the system. After loading, the parsing program decompiles the code 
state after loading the program, so the program is not running during static analysis. This analysis 
method can intuitively see the results of anti-assembly, and analyze its specific functions by reading 
the source code. However, in some special cases, when you need to obtain some runtime parameter 
values, it is powerless. It can only analyze its logic pseudo code, and cannot get the running result, 
which is the disadvantage of this analysis method.

2.2.2. Dynamic analysis. Dynamic analysis [3] is also to load the program to be analyzed into the 
memory, and load program of the simulation system. It loads the program, and then stays at the 
program entry point, waiting for further execution. In the beginning, dynamic analysis and static 
analysis are consistent, but the program can be further executed. During execution, you can see the 
execution data of program, that is, you can see the process of the system running program through the 
interface. The disadvantage of dynamic analysis is that when analyzing particularly complex program 
logic, after a long period of debugging, the displayed program logic may be confused.

2.3. Anti-debugging
The anti-debugging method [4] is a very common anti-analysis method, that is, adding an anti- 
debugging function to the program logic. The program determines whether the program is debugging 
or not through various methods, and determines the direction of the program flow through the 
judgment result. The exit function is executed when debugging, and the original function is executed 
when the program is judged not to be debugged. The packer designed in this article adds the most 
basic anti-debugging method. When the program is about to complete the interface initialization, the 
judgment of PEB is added. The structure value is being debugged to decide whether to exit or continue 
execution. Through this anti-debugging method, the safety of the packer program is guaranteed to a 
certain extent [5].

3. Scheme design and implementation

3.1. Process
The design idea of this packaging tool is to write a custom command interpreter in assembly language. 
Based on the virtual shell technology, QT5 framework is used as the front-end page frame to design a 
virtual shell packer. After the protection code is decompiled, the source code logic becomes chaotic, 
which greatly increases the difficulty of cracking, thus realizing the protection of the software. The 
basic workflow is shown in Figure 1:

3.2. Custom instruction set and interpreter

3.2.1. Custom instruction set. Through the analysis of normal assembly instructions, a custom 
language format is designed. Based on this format, forty-eight instructions are selected to generate
their corresponding custom instruction sets, which can cover most instructions used by the programs. The following code shows part of the custom instruction set for this design.

| Code | Description                  |
|------|------------------------------|
| 1.   | \
| 2.   | \
| 3.   | \
| 4.   | \
| 5.   | \

```c
#define I_COND_JMP_SHORT 0x00
#define I_COND_JMP_LONG 0x01
#define I_JECX 0x02
#define I_CALL_REL 0x03
#define I_VM_END 0x04
```

3.2.2. The interpreter. The interpreter is the core of this packer, and is responsible for the instruction conversion between the custom instruction set and the normal instruction set. Without the interpreter, the function of the packer will not work properly. The functions of the interpreter can be divided into three: (1) decryption of encrypted instructions; (2) recognition of decrypted instructions; (3) functions performed by the simulated instruction function.

Through the analysis of the execution process, the function executed by the simulation instruction function must convert and simulate the instruction function of the custom instruction set. When designing a function that simulates the execution of an instruction function, you need to understand the parameter requirements of each instruction, so that the protected section code of the protected program can be executed like a normal assembly instruction.

3.3. Packer implementation

3.3.1 Interface design. The interpreter interface is designed through the QT framework. The third line of the main interface is "add section" and "packing button". The button of the additional section is used to open the code section for the user to fill in the protection, as shown in Figure 2 Show:

3.3.2 Instruction conversion. It can be seen from the foregoing that the instruction code and data of the normal assembly instructions need to be obtained first, and then the custom analysis is performed on the normal assembly instructions to obtain a custom structure, that is, the normal assembly instructions are analyzed using the hde_disasm function included in the HDE. The parsed results are filled into this
3.3.3. Command encryption. The previously generated custom instructions can be parsed through multiple instructions. Therefore, the instructions need to be encrypted and protected. Through a series of reversible operation instructions, the converted custom instruction data is operated to protect the converted custom instruction. The following code uses common partial reversible arithmetic instructions.

```
//XOR val 0x34 val
//SUB val 0x2C val
//ADD val 0x04 val
//XOR CL 0x32 0xC1
//SUB CL 0x2A 0xC1
```

3.3.4. Instruction decryption. Since it is necessary to decrypt the operation objects that need to be decrypted, an array of reversible operation instructions are used to store the records during the encryption process, and the same operation is stored. At the same time, the data is written to the protected program, which can be directly called during decryption. The following code is the storage encryption operation code.

```
memmove(_vm_poly_dec, _vm_poly_enc, sizeof(_vm_poly_enc));
*(DWORD*)(_vm_poly_enc + 0x65) = 0x18244433;
*(DWORD*)(_vm_poly_dec + 11) = 0x18244433;
```

3.3.5. Add anti-debugging. After the basic functions of this packer are completed, in order to further protect the security of the program source code of this tool, an anti-debugging function for the program code is added during the window initialization process, that is, to check the BeingDebuged field of the program to determine whether the program is being debugged. To a certain extent, the anti-debugging protection of the packed program is realized. The following code is the code to implement the anti-debugging function.

```
bool isDebug()
{
    asm {
        ; //Get the content of PEB
        mov eax, fs:[0x30]
        ; Get the PEB.BeingDebuged
        movzx eax, byte ptr[eax + 2]
    }
    return (eax == 0);
}
```
4. Functional test

4.1. Perform pre-packing procedures
Figure 3 is a simple protected program execution results before packing:

![Figure 3. Execution results before packing](image)

4.2. Test whether anti-debugging is achieved
The function of anti-debugging is to prevent illegal debugging of this packer through dynamic analysis tools [6]. The packer recognizes that the program is being debugged and executes the exit program code, indicating that the anti-debugging function added by the packer realizes the protection of the program.

4.3. Static testing and analysis
The static analysis tool IDA analyses the disassembly results of the program before and after packing protection, analyses the packing function from the static reverse level, and finds that the unpacked program can be analysed by the pseudo code of the program, and the code of the protected program after the packer is very obvious. Figure 4 shows the pseudo code obtained by IDA analysis before and after the program is not protected. In this condition the corresponding pseudo code cannot be obtained by IDA analysis:

![Figure 4. Static analysis comparison of the program before and after packing protection](image)

4.4. Dynamic testing and analysis
Dynamic debugging is to debug the assembly code, understand the execution flow of the program through interpretation of the assembly code, and also know which functions and parameters are used for debugging. As shown in Figure 5, the dynamic analysis results are clearly visible before the program is not protected. After using this packer, the code of the packing area has been changed.

![Figure 5. Dynamic analysis results before](image)
When the code is executed, it will jump into the interpreter to explain and load the previously encrypted instructions to ensure the normal execution of the function, but the analyst cannot obtain the functionality. Thus achieving the purpose of dynamic debugging, the code of the instruction shown in Figure 6.

![Figure 6. After program pack protection](image)

5. Result analysis and conclusion

This paper proposes a software protection method and implementation based on virtual shell technology. The macro process of shell protection is to process the source code through various means so that it cannot be directly statically analyzed into normal source code. The original instructions are converted into custom instructions, and interpreter is implemented, and the customized instruction is parsed through the interpreter. The analysis becomes the original instruction function, which ensures the logic of the original program. The virtual shell protection method’s core key lies in the design of instructions.

Through testing, the program added by this packer can perform the original function normally. The results of static analysis and dynamic debugging analysis prove that packer protection has an obvious protective effect on the software program in preventing source code leakage.

Acknowledgments

In this paper, the research was sponsored by the Key curriculum construction project of Xi'an Eurasia University (Project No. 2019KC003)

References

[1] Feng, C.Y, Fu, Y.F, Cui Y.P. (2015), Analysis of PE file adding section program. Electronic Science and Technology. J, 02(05): 561-568.
[2] Qi, Y.M. (2019), Protection and Recovery Technology of Computer Software System. Information Recording Materials. J, (5): 163-164.
[3] Tian, N, Pu, J. (2017), A review of software dynamic defense technology research. Cyberspace Security. J, (8): 38-42.
[4] Wang X.C, (2017), QT Graphical Interface Programming and Optimization. Computer Fans. J, (18): 101-104.
[5] Huang, D.K, Gao, W. (2018), Discussion on the protection methods of computer software. China's Strategic Emerging Industries. J. (24): 120.
[6] Xie, X, Xiang, F, (2018), A virtual machine software protection method based on enhanced scheduler. Computer Applications and Software. J, 35(11): 8-15+48.