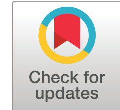
Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.038970



ARTICLE

A Smart Obfuscation Approach to Protect Software in Cloud

Lei Yu[1] and Yucong Duan[2,*]

[1]Department of Computer Science, Inner Mongolia University, Hohhot, China

[2]Department of Computer Science, Hainan University, Haikou, China

*Corresponding Author: Yucong Duan. Email: duanyucong@hotmail.com

Received: 05 January 2023 Accepted: 10 April 2023 Published: 08 October 2023

ABSTRACT

Cloud computing and edge computing brought more software, which also brought a new danger of malicious software attacks. Data synchronization mechanisms of software can further help reverse data modifications. Based on the mechanisms, attackers can cover themselves behind the network and modify data undetected. Related knowledge of software reverse engineering can be organized as rules to accelerate the attacks, when attackers intrude cloud server to access the source or binary codes. Therefore, we proposed a novel method to resist this kind of reverse engineering by breaking these rules. Our method is based on software obfuscations and encryptions to enhance the security of distributed software and cloud services in the 5G era. Our method is capable of (1) replacing the original assembly codes of the protected program with equivalent assembly instructions in an iteration way, (2) obfuscating the control flow of the protected program to confuse attackers meanwhile keeps the program producing the same outputs, (3) encrypting data to confuse attackers. In addition, the approach can periodically and automatically modify the protected software binary codes, and the binary codes of the protected software are encrypted to resist static analysis and dynamic analysis. Furthermore, a simplified virtual machine is implemented to make the protected codes unreadable to attackers. Cloud game is one of the specific scenarios which needs low latency and strong data consistency. Cheat engine, Ollydbg, and Interactive Disassembler Professional (IDA) are used prevalently for games. Our improved methods can protect the software from the most vulnerable aspects. The improved dynamic code swapping and the simplified virtual machine technologies for cloud games are the main innovations. We inductively learned that our methods have been working well according to the security mechanisms and time complexity analysis. Experiments show that hidden dangers can be eliminated with efficient methods: Execution time and file sizes of the target codes can be multiple times than that of the original program codes which depend on specific program functions.

KEYWORDS

Obfuscation; self-modification; encryption; edge computing

1 Introduction

The technologies of cloud computing, edge computing, and 5G have been becoming mature and bringing more software, which also brings a new danger of malicious software attacks. Data

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

synchronization mechanisms of software can further help reverse data modifications. Based on the mechanisms, attackers can cover themselves behind the network and modify data undetected. Related knowledge of software reverse engineering can be organized as rules to accelerate the attacks when attackers intrude cloud server to access the source or binary codes. It is a specific scenario that needs specific methods to protect software. Cloud game is one of the specific scenarios which needs low latency and strong data consistency. Cheat engine, Ollydbg, and IDA are used prevalently for games that are even protected by some security mechanisms. There are multiple ways to obtain sensitive data or target codes, leading to single protection failure in most cases. How to hinder attackers to bypass protection methods are complex.

Though software encryptions and decryptions can be used for hiding data, they are computationally heavy for some distributed software (especially game) protections, thus few of them are applicable. In contrast, software obfuscations and tamper-proofs are the most common techniques. To protect software, software obfuscations aim at removing useful information. However, only protecting codes is not enough because data attacks account for at least half of the total attacks. Data obfuscations are used to hide the values of program variables. Some researchers did not consider game environments, thus they lack efficiency compatible with gameplay. Some researchers used deep learning to enhance software security. However, they only focus single aspect to protect software security. Our proposed framework filled the above gaps by considering multiple protection ways to protect both game data and codes.

We proposed corresponding methods to resist this kind of reverse engineering by breaking these rules. Our countermeasures are based on software obfuscations and encryptions to enhance the security of distributed software and cloud services. To protect the software from the most vulnerable aspects, the improved dynamic code swapping and the simplified virtual machine technologies are our features because no researcher used them in cloud games. We inductively learned that our methods have been working well by security mechanisms and time complexity analysis. Improved dynamic code swapping and simplified virtual machine technologies are easy to implement, by which readers can design their sensitive functions. Technical details and main codes are shown in our methods, by which readers can effortlessly implement them in any programming language. How to break the knowledge of attackers to resist reverse data manipulations in distributed software? There are challenges for current work in reality:

1. Attackers have enough software reverse tools for binary source codes. Recent commercial and free software reverse tools are powerful but not smart enough, because a deep learning system can hardly fulfill tasks of reverse engineering without human knowledge [1]. However, they still save much time and effort for attackers.
2. Attackers have enough theoretical knowledge about software reverse engineering for non-obfuscated original binary source codes. We assume they at least have basic theories as the paper [2] presented.

To tackle the above challenges, we proposed a novel Self-Modified Encrypted Obfuscation approach called SMEO to protect the software and games. The main contributions of this paper are shown as follows:

1. Hidden dangers of distributed software, especially games, are revealed. Detailed countermeasures for reverse data modifications are illustrated: Our overall idea to avoid reverse data modifications is that reversely do the reverse things.

2. A novel obfuscation approach (SMEO) is proposed. The approach can periodically and automatically modify the protected software binary codes, and the protected software binary codes are always encrypted to resist static analysis and dynamic analysis. Furthermore, a simplified virtual machine is implemented to make the protected codes unreadable. Especially, SMEO is efficient for computer games.

This paper is organized as the followings: Section 2 discussed software analysis, obfuscations, encryption, and tamper-proof. A protection methodology against reverse data modifications was proposed in Section 3, which includes data encryptions, static and dynamic obfuscations, and tamper-proof methods. Section 4 showed the experiment results and proved that our method is applicable according to the evaluations of parameters of computer performance.

2 Related Work

Related research includes obfuscations, tamper-proofs, encryptions, and decryptions. Though software encryptions and decryptions can be used for hiding data, they are computationally heavy for some distributed software (especially game) protections, thus few of them are applicable. In contrast, software obfuscations and tamper-proofs are the most common techniques.

To protect software, software obfuscations aim at removing useful information. Foket et al. [3] proposed to combine five transformations that obfuscate the type hierarchy of Java applications and eliminate much of the type information. Albrecht et al. [4] provided constructions of multilinear groups equipped with natural hard problems from indistinguishability obfuscation, and homomorphic encryption. Utilizing indistinguishability obfuscation, a multi-hop unidirectional proxy re-encryption scheme against chosen-ciphertext attacks (CCA) is proposed [5]. Dachman-Soled et al. [6] proposed a modular approach based on program obfuscation and presented a compiler. However, only protecting codes is not enough because data attacks account for at least half of the total attacks.

Data obfuscations are used to conceal the actual values of program variables. An extension of XOR-Masking is presented [7] where the mask is an opaque constant. An obfuscation approach is proposed [8] by look-up tables as reconfigurable logic to replace the carefully selected gates. A low-cost functional obfuscation methodology is proposed [9] through the employment of a robust Intellectual property locking technique. Zhang et al. [10] proposed an obfuscation scheme achieved through varying finite field constructions and primitive element representations. A set of quantitatively evaluable metrics are proposed [11], and Olney et al. [12] proposed a tunable obfuscation approach from typical bitstream attacks while enabling designers to trade off security with acceptable overhead.

Some researchers considered location-based obfuscation: A framework with geo-obfuscation is proposed [13] to protect users' locations during task assignments; A Proactive Topology Obfuscation (ProTO) system that adopts a detect-then-obfuscate framework is proposed [14]; A location obfuscation method [15] is proposed which is robust to privacy inferences by the service provider regarding route source and destination.

Traditional cryptographic primitives for data integrity protection cannot be directly used because they cannot ensure security in the case of collusion between the cloud server and the local host [16]. Sutton et al. [17] proposed a Linked Data-based method to create tamper-proof audit logs. Fully

homomorphic encryption allows computations on encrypted data without the need for decryption and it provides privacy in various applications [18].

Zhao et al. [19] used a deep neural network and semantic information of the disassembled binary to predict if the program has been obfuscated. Bui et al. [20] introduced a deep network architecture to compute temporal content hashes (TCHs) from audio-visual streams. TCHs are sensitive to accidental or malicious content modification (tampering). Shayan et al. [21] introduced the use of a dummy valve as a security primitive to obfuscate bioassay implementations and presented design rules and security metrics to design and measure obfuscation. By investigating UI obfuscation, Zhou et al. [22] pointed out the weaknesses in existing automated UI analysis methods and designed 9 UI obfuscation approaches. An algorithm for reversible data hiding is proposed [23], which is further work for the paper [24]. A code division multiplexing (CDM) algorithm-based reversible data hiding (RDH) scheme is presented [24].

3 Consolidate Distributed Software by SMEO

Cheat engine, Ollydbg, and IDA are used prevalently for games that are even protected by some security mechanisms. There are multiple ways to obtain sensitive data or target codes, leading to single protection failure in most cases. How to hinder attackers to bypass protection methods are complex. Therefore, we proposed several methods, especially for games. Improved dynamic code swapping and simplified virtual machine technologies are our features, by which readers can design their sensitive functions. Technical details and main codes are shown in our methods, by which readers can effortlessly implement them in any programming language. Several methods are proposed in this section to confuse attackers and prevent data reverse modifications.

3.1 Data Homomorphism Encrypt Method

Software data should be encrypted to confuse attackers, but the balance between computation performances and security levels needs to be considered, especially in games. RSA (stands for Ron Rivest, Adi Shamir and Leonard Adleman) algorithm only supports multiplication homomorphism, while the Paillier algorithm is a heavy homomorphism encryption method that supports both multiplication homomorphism and addition homomorphism. In most cases, the theory of finite field gives help, but finite field Galois (2^8) with the modular operation of polynomials is heavy too. Following lightweight homomorphism encryption formulas are suitable for integer encryption in games, which is used by SMEO.

$$\begin{cases} \text{Encrypt (plain)} = \text{cipher} = \text{randoms} \times \text{N} + \text{plain} \\ \text{Decrypt (cipher)} = \text{plain} = \text{cipher\% N} \\ \text{Add (cipher1, cipher2)} = \text{cipher1} + \text{cipher2} \\ \text{Mul (cipher2, cipher2)} = \text{cipher1} \times \text{cipher2} \end{cases} \quad (1)$$

Variables (plain and cipher) are integers. N is a multiplication of two neighbor prime numbers, and randoms are different values in different clients to further confuse attackers, which can be used for hidden data.

Data sequence should be reconstructed by folding (increasing the dimension of the array), unfolding (decreasing the dimension of the array), or other identity transformations. The identity transformations are based on the theory of the primitive root of a prime integer. If r is a primitive root of a prime integer p, then sequence {r^0% p, r^1% p, r^2% p, r^3% p, ..., r^{p-1}% p} is a substitution of sequence {1, 2, 3, ...,}. Another lightweight homomorphism function can be used to rearrange a data sequence:

$$\text{homo (index)} = (\text{index} \times \text{M})\ \%\ \text{N} \tag{2}$$

N is the length of the data sequence, and M is a relative prime number to N. For example, N = 5 and M = 3:

$$\begin{Bmatrix} \text{index}: 0, 1, 2, 3, 4 \\ \text{data}: \ 1, 2, 3, 4, 5 \end{Bmatrix} \xrightarrow{\text{homo (index)}} \begin{Bmatrix} \text{index}: 0, 1, 2, 3, 4 \\ \text{data}: \ 1, 4, 2, 5, 3 \end{Bmatrix} \tag{3}$$

3.2 Static Obfuscation Method

Static obfuscation can be used against static analysis. For example, SMEO uses variables A, B, C, D, E, F, i, and ii to simplify a program where irrelevant variables are removed and substituted:

Example

```
class CombatUnit: GameObject
        void Serialize (Stream & stream)
                // local variables A,B,C,D,E,F,i,ii
                A = B // auto& vars = dataType- > GetVars()
                while i < 20 // foreach i in vars.size()
                        if ii ! = 0       // if ((1 < < i) & vars_BitField) ! = 0
                                E = C + D     // void ∗ data = this + var.offset
                                switch F                // switch var.primitiveType
                                        case 0:         //case P_int:
                                                function()      // stream.Serialize(∗(int∗)data)
                                        case 1:         //case P_float:
                                                function()      // stream.Serialize(∗(float ∗)data)
                        i++
```

Next, SMEO obfuscates the control flow of the above program to confuse attackers meanwhile keeps the program producing the same outputs.

Protection Program in SMEO: Obfuscate the Control Flow

```
Partial class CombatUnit: GameObject
 int roller[] = {2,3,5,7,11}
 int n = roller.size
 int ∗ alias1, alias2 = &n
 void Serialize (Stream & stream)
        // local variables A,B,C,D,E,F,i,ii
        int next, rot = 0
```

(Continued)

Protection Program in SMEO: Obfuscate the Control Flow (continued)

```
        bool run = true
        while(run)
                switch (next)
                        case 0: A = B; next = roller[(0 + rot)%n] − 1 = 1; break;
                        case 1: if (i < 20) next = 2; else run = false; break;
                        case 2: if (ii != 0) next = ∗alias1-2 = 3; else next = 7; break;
                        case 3: E = C + D; next = roller[(3 + rot)%n]-roller[(1 + rot)%n] = 4;break;
                        case 4: if (F == 0) next = 5; else next = ∗alias1 + 1 = 6; break;
                        case 5: function(); next = roller[(roller[(1 + rot)%n] + rot)%n] = 7;break;
                        case 6: if (F == 1) next = 5; else next = 7; break;
                        case 7: x = a random positive integer; G = x∗(x + 1)% 2;
                                if (G) next = 2; else next = 9; break;
                        case 8: x = a random positive integer; P = a prime number;
                                G = pow(x, P − 1)% P; if (G) next = 9; else next = 4; break;
                        case 9: i++; next = 1; break;
                        case 10: ∗alias1 = 3; ∗alias2 = 6; next = roller[(3 + rot)%n]∗2; break;
                Rotation ()
                rot = (rot + 1)% roller.size
 void Rotation ()
                int tmp = roller[n − 1]
                for (i = n – 2; i >= 0; i–)
                     roller[i + 1] = roller[i]
                roller[0] = tmp
```

To confuse attackers, opaque predicates are used in case 7 and case 8, where it is inevitable that next = 9. The reasons are that an odd number multiplied by an even number produces an even number in case 7; the Fermat theorem is used in case 8. In addition, redundant case 10 is inserted which will never be executed in any case. To increase dynamic data, function Rotation() is used to move each element in the roller to the right, meanwhile "rot" is used to keep the "next" calculation not changed. Alias analysis often consumes attackers lots of time, thus alias should be used in the control flow.

If obfuscation occurs in assembly codes, the following target instructions in Table 1 can be replaced by equivalent instructions in an iteration way. For the example of "push eax", the game developers can use semantic style 2.3 to replace it, then replace sub esp, 4 with semantic style 1.

Table 1: Equivalent instructions

Semantic style	Semantic	Target instructions	Equivalent instructions
1	Change esp by subtraction	sub esp, 4	push register mov register, esp xchg [esp], register pop esp
2	Push the value of a	push register	push 0x0

(Continued)

Table 1 (continued)

Semantic style	Semantic	Target instructions	Equivalent instructions

	register to the stack		mov [esp], register
			lea esp, [esp-4]
			mov [esp], register
			sub esp, 4
			mov [esp], register

3	Move an immediate number to a register	mov register, 0xD	0x0: call 0x05 0x05: pop register 0x06: xor register, 0x08
4	Jump to a memory address	jmp 0x66	push 0x66 Ret 0x00: call 0x66 0x05: interference codes or data ... 0x66: add esp, 4 0x00: call function 0x05: interference codes or data ... 0x66: nop function: add [esp], 0x61 function+n: ret

Example	Push the value of eax to the stack	push eax	push eax
			mov eax, esp
			xchg [esp], eax
			pop esp
			mov [esp], eax

3.3 Dynamic Obfuscation Method

Return-Oriented Programming (ROP) can be used to change control flow. Sometimes disassembly codes cannot be parsed correctly by linear scanning of machine codes when data and instructions are mixed. Therefore, recursive methods are used to parse machine codes into assembly codes. However, return addresses of functions in the stack can be modified in runtime (e.g., inc dword ptr [esp+n]), leading to failures of static analysis, which are called self-modifying codes. This kind of technique can also be applied to anti-cheat modules. We used dynamic obfuscation and self-modification to destroy the prior knowledge of attackers that:

(1) Program instructions must locate in the same Relative Virtual Address (RVA).
(2) There is no data block in code segments.

We proposed a dynamic obfuscation method combined with a simplified virtual machine. The dynamic obfuscation method can modify instruction addresses and encrypt code blocks. The rationale

of the dynamic obfuscation method is that 3 xors can replace the position of two objects (A, B). Furthermore, 6 xors can restore the position of the two objects (A, B):

$$\genfrac{}{}{0pt}{}{A}{B}\downarrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{A}{A\oplus \mathrm{B}}\uparrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{A\oplus \mathrm{A}\oplus \mathrm{B}}{A\oplus \mathrm{B}}\downarrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{B}{A}\uparrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{B\oplus \mathrm{A}}{A}\downarrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{B\oplus \mathrm{A}}{B}\uparrow\oplus \Rightarrow \genfrac{}{}{0pt}{}{A}{B} \tag{4}$$

Fig. 1 shows the process of our dynamic obfuscation. Space 0–5 are memory address spaces that store code blocks. Only one space stores plain code blocks and other spaces store mutation code blocks at any time. Furthermore, the code blocks are executed in a linear sequence, but not in linear address space to destroy the expectation of attackers. Fig. 1 shows space travel if a program contains code blocks A, B, C, D, E, and F, which should be executed in order. A mutant code block (M0, M1, M2, M3, M4, and M5) is either a plain code block or a mutation code block. Codes in the plain code block must be readable and executable by a CPU, while codes in the mutation code blocks must be unreadable to attackers.

In the first step, M0 in space 0 must be readable and executable by the CPU, thus M0 is code block A. Meanwhile, Space 1, Space 2, Space 3, Space 4, and Space 5 should store the mutation code blocks, which means M1, M2, M3, M4, and M5 should be the mutation code blocks. In step 1.2, each mutation code block in the lower part (M3, M4, M5) is xor by each mutation code block in the upper part (M0, M1, M2).

In the second step, M0 ⊕ M3 in Space 3 must be readable and executable by the CPU, thus M0 ⊕ M3 is code block B. Meanwhile, Space 0, Space 1, Space 2, Space 4, and Space 5 should store the mutation code blocks. In step 2.2, each mutation code block in the upper part (M0, M1, M2) is xor by each mutation code block in the lower part (M0 ⊕ M3, M1 ⊕ M4, M2 ⊕ M5).

In the third step, M4 in space 1 must be readable and executable by the CPU, thus M4 is code block C. Meanwhile, Space 0, Space 2, Space 3, Space 4, and Space 5 should store the mutation code blocks. In step 3.2, each mutation code block in the lower part (M0 ⊕ M3, M1 ⊕ M4, M2 ⊕ M5) is xor by each mutation code block in the upper part (M3, M4, M5).

Likewise, code blocks D, E, and F are executed in a crossed order, and the program is executed one time until F is finished. Therefore, one execution is a circle containing 6 steps. The following equations reveal a map of the code blocks and the mutation code blocks.

$$\begin{cases} M_0 = A \\ M_0 \oplus M_3 = B \\ M_4 = C \\ M_1 = D \\ M_2 \oplus M_5 = E \\ M_5 = F \end{cases} \Rightarrow \begin{cases} M_0 = A \\ M_1 = D \\ M_2 = E \oplus F \\ M_3 = A \oplus B \\ M_4 = C \\ M_5 = F \end{cases} \tag{5}$$

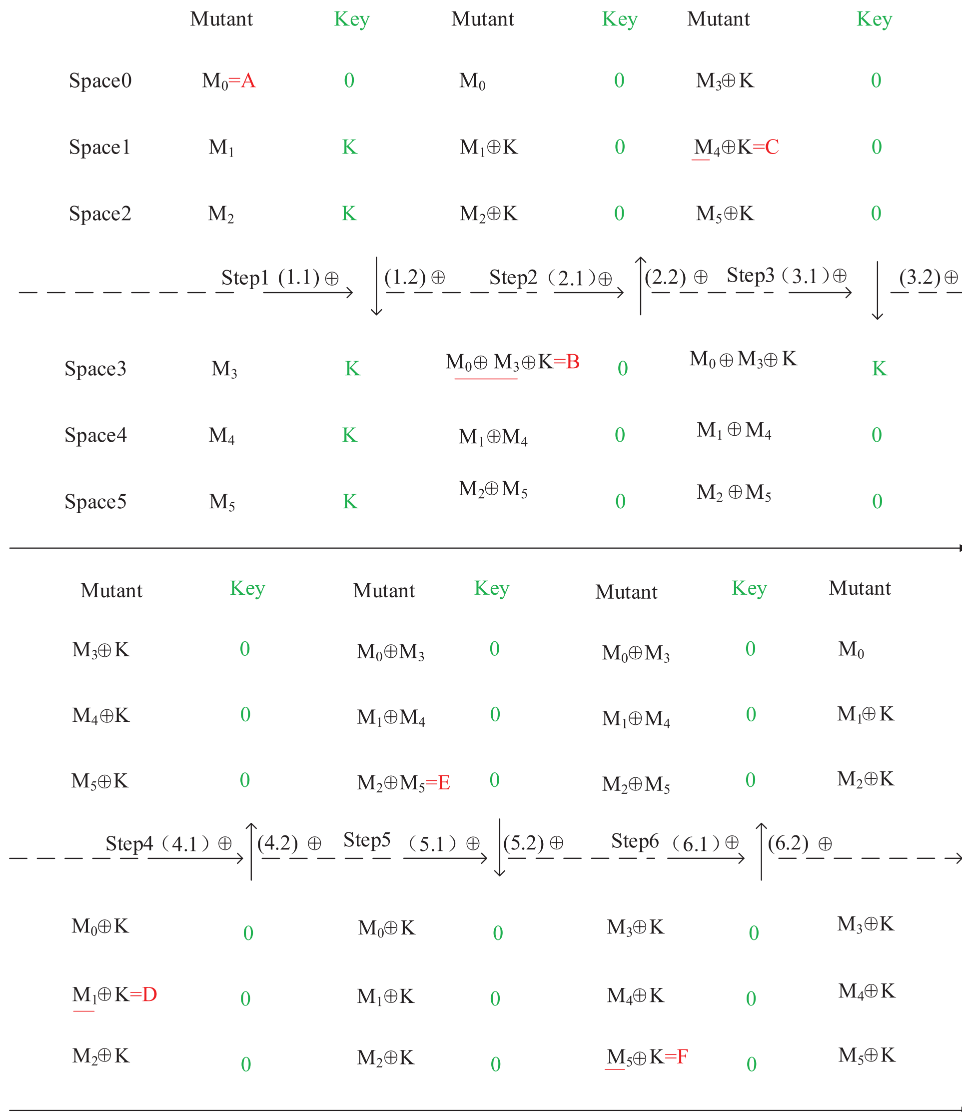


Figure 1: Dynamic obfuscation with encryptions

The following program shows how SMEO encrypts the code blocks and changes the memory addresses of the code blocks.

Protection Program in SMEO: Self Modification

```
void xor (int source_addr, int dest_addr, int len, int key)
        for (i = 0; i < len; i++)
                *dest_addr ^ = *source_addr
                *dest_addr ^ = key
                dest_addr++
```

(Continued)

Protection Program in SMEO: Self Modification (continued)

```
                    source_addr++
void swap_code (int source_addr, int dest_addr, int len)
          for (i = 0; i < len; i++)
                    int t = *source_addr
                    *source_addr = *dest_addr
                    *dest_addr = t
```

According to the above equations, the program is stored in the memory address in the order of A, D, $E \oplus F$, $A \oplus B$, C, and F. However, it is still obvious that code blocks A, D, C, and F are exposed to attackers. Therefore, the green key in Fig. 1 is used for xor with any exposed code block, so that only one code block is exposed at one time. In practice, any plain code block will be xor with k in each step x.1. Notice that the result of 0 xors with arbitrary codes will change nothing. Right now the exposed plain code block is Achilles' heel, thus the following simplified virtual machine can be used for boots.

Protection Program in SMEO: Virtual Machine

```
Partial class CombatUnit: GameObject
          void Serialize (Stream & stream) {
                    int stack [20], stack_pointer = 0;
                    void * prologue[] = {
                    //A = B
                    &&pushl, &A, &&pushr, &B, &&store,
                    // if (i < 20)
                    &&if_i_less_20,
                              // if ii != 0
                              &&if_ii_ne_0, &prologue [17]
                                        //E = C + D
                                        &&pushl, &E, &&pushr, &C, &&pushr, &D, &&add, &&store,
                                        //switch F
                                        &&if_F_e_0_1,
                                                  //case 0:
                                                             // function()
                                                  //case 1:
                                                             //function()
                                        //i++
                                        &&inc_i,
                                        &&jump, &prologue [0]
                    };
                    void ** prog_counter = (void**) & prologue;
                    goto ** prog_counter++;
                    if_i_less_20: if (i < 20) goto ** prog_counter ++; else return;
                    if_ii_ne_0: if (ii != 0) prog_counter ++; else prog_counter = * prog_counter ;
                                      goto ** prog_counter ++;
                    if_F_e_0_1: if (F == 0 || F == 1) function(); goto ** prog_counter ++;
                    pushl: stack[stack_pointer ++] = (int) * prog_counter;
                              prog_counter ++; goto ** prog_counter ++;
```

(Continued)

Protection Program in SMEO: Virtual Machine (continued)

```
                    pushr: stack[stack_pointer ++] = * (int*) * prog_counter;
                               prog_counter ++; goto ** prog_counter ++;
                     store: stack[stack_pointer ++] = * ((int*) stack[stack_pointer-2])= stack[stack_
pointer −1];
                                  stack_pointer-=2; goto ** prog_counter ++;
                    inc_i: i++; goto **prog_counter++;
                    add: stack[stack_pointer-2] + = stack[stack_pointer-1]; stack_pointer–;
                         goto ** prog_counter ++;
                    jump: prog_counter = * prog_counter ; goto ** prog_counter ++;
          }
```

A data stack is built for math operations, and data can be retrieved by a stack_pointer. In addition, a program instruction counter of the virtual machine is simulated by prog_counter.

3.4 Tamper Proofing Method

Attackers can do bad things even without recompiling the software that is written in Python once the attackers intrude cloud servers. The second wall against attackers is tamper-proofing. The differences between obfuscation and tamper-proofing are that tamper-proofing assumes attackers have broken through the obfuscation, and they are modifying the program codes. Check summing and program-monitor are most frequently used for tamper proofing.

A monitor program can detect malicious modifications to an original program, but can we design a monitor program that can detect malicious modifications to itself? A complete self-monitor program is related to a Russell paradox or a halting problem, thus we use a protection net as Fig. 2 shows.

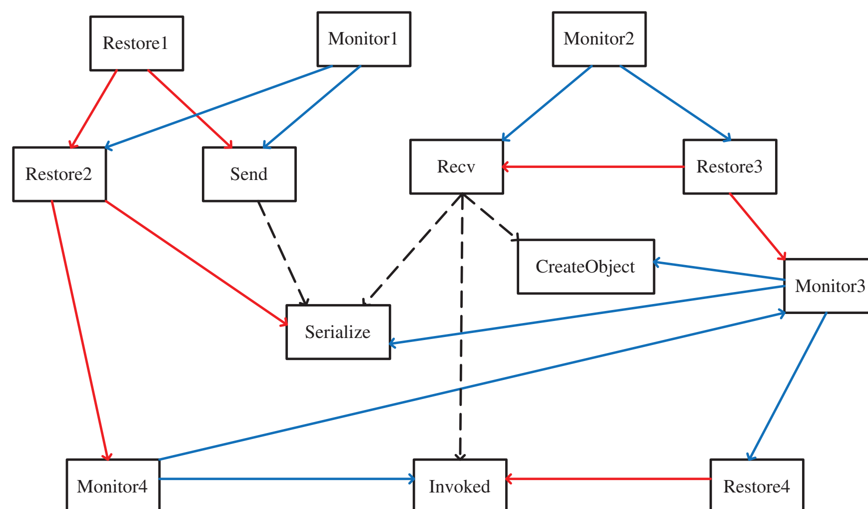


Figure 2: Protection net

This protection net of SMEO can protect program codes, where dot lines are invocation relations. Monitor programs (e.g., Monitor1) are used to detect malicious modifications to original program functions (e.g., Send), and restore programs (e.g., Restore1) are used to restore tampered functions to the original functions. If two monitor programs are used to monitor each other, the following program

should be used. Therefore, the Russell paradox is avoided by skipping the address of "compare_hash".

Protection Program in SMEO: Self Monitor

```
int Integrity (int address_start, int address_end, int odd)
        int hash = 0
        while address_start <= address_end
                if address_start == address_compare_hash
                        continue
                hash = odd * (*address_start + hash)
                address_start ++
        return hash
int SelfMonitor (int address_start, int address_end, int odd)
        if Integrity(address_start, address_end, odd) != hash_value
                Restore (address_start, address_end)
                return 0
```

3.5 Bypassing Injection Prevention Mechanism and Countermeasures in Practice

Attackers can easily inject their DLLs (Dynamic Link Library) into software processes if the software has not or has weak prevention mechanisms to protect the integrity. For example, the software can hook function NtOpenFile in ntdll.dll to detect DLL injection. However, anti-load library protections can be bypassed by restoring several bytes of the original NtOpenFile, which is a "jmp" instruction. In addition to LoadLibrary detection, thread creation detections can be used. For example, we can use DLL_THREAD_ATTACH in DllMain to call the function NtQueryInformationThread in ntdll.dll to get the start address of the newly created thread. Therefore, several bytes should be patched with the instruction "jmp" to skip thread creation detection.

In addition, some anti-reverse engineering methods can be used to enhance prevention mechanisms at the system level. Game developers can use anti-debug methods to protect important codes and data. The current privilege of the CPU can be changed by instructions "int 3" (ring 3 to ring 0) and "iret" (ring 0 to ring 3). For anti-debug, game developers can insert the instruction "int 3" into programs and put important codes in exception functions to avoid execution when the programs are being debugged because "int 3" will be considered a breakpoint by a debugger. In addition, "debugging" and "NtGlobalFlags" in PEB (Process Environment Block) indicate the game programs are being debugged.

4 Experiments

4.1 Evaluations

We developed three games to evaluate and analyze the performances of the protection methods. Two computers with 2.5 GHZ intel i7 CPU and 16 G Memory, are used for experiments and evaluations. The results of our obfuscation and tamper-proof methods are shown, which assures high-level security and high-speed computations (xor operations) because xor operations are fast in encryption methods.

What is changed for software with and without protections? There are two factors changed, which are file sizes and process execution time. Table 2 shows the comparison of file sizes with and without protection.

Table 2: File size

Protected software	Without protection (MB)	With protection (MB)
game1	26	47
game2	135	313
game3	412	967

SMEO expands the file sizes of the protected software by changing assembly codes and patching functions, based on static obfuscations. In practice, a smaller file size means that fewer codes are executed and less time is consumed. In contrast, a larger file size is easier to confuse attackers, but it may affect game performance. Therefore, game security experts need to have a good balance. Table 3 shows the comparison of execution time with and without protection.

Table 3: Execution time of the target codes

Protected software	Without protection (microsecond)	With protection (microsecond)
game1	0.6	121
game2	2.3	269
game3	6.2	818

Target codes are core codes that need to be hidden or protected. How to choose target codes is another technique. SMEO increases the execution time of protected software by dynamic obfuscation and self-monitor. Increment in the execution time of the target codes sometimes affects game performance but does not decrease players' game experiences.

How do the actions of game players affect game performances in scenarios with and without protections? Different player actions and different protected code blocks affect the computation performances, which will be comprehensively analyzed for different kinds (Massively Multiplayer Online, Simulations, Adventure, Real-Time Strategy, First Person Shooters, Sports, and Role-Playing) of computer games to help game developers validate performances in various online games.

4.2 Time Complexity

The time complexity of our methods is discussed comprehensively. The inputs of data encryption methods are program variables, and the inputs of code encryption methods are program codes. The time complexity of the Data Homomorphism Encrypt Method depends on the number of target variables in the protected program, while the data encryption operation does not depend on the number of target variables, thus the total cost is O(n), meaning that time increases linearly when the number of target variables increases n times. The time complexity of the Static Obfuscation Method depends on the number of target codes for the function in the protected program, which (1) manipulates the control flow; (2) expands the machine codes. The first part depends on the target function and the roller size. Its principle is a specific-designed rearrange, thus it is O(n). The second part depends on the

epochs (suppose m, which fortunately can be set by program developers) of instruction substitutions from Table 1, thus it is $O(n^m)$. The time complexity of the Dynamic Obfuscation Method depends on the number of target codes for the function in the protected program, which manipulates the control flow. The main loop of Self Modification is six iterations for code blocks and O(n) for swapping codes in each iteration, thus it is O(n). The time complexity of the Virtual Machine depends on the target function and its principle is instruction expansion. The increased time can be ignored when the number of target codes increases n times in most cases, thus it is O(1).

4.3 Discussions

In most cases, attackers just manipulate high-level data, and no need to destroy the integrity of network data synchronization. Especially, online multi-player games are the distributed software systems, thus problems of data synchronization are considered like ordinary distributed software systems. Game data synchronization maintains the consistency and uniformity of game data across all player clients and game servers. When one client modifies data by the game operations of a player, other clients or game servers (if P2P is not used) must be notified of the changes. The process of data synchronization also can be exploited by reverse engineering of network protocol of distributed software systems. Taint analysis can be used to mark instructions and data to obtain repeating patterns for partitioning message fields of a network protocol. Furthermore, a protocol state machine can be revealed by a sufficient quantity of successive messages, once message fields and message types are identified.

Knowing whether Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) or modified transport protocols (e.g., sequential-assurance UDP) that software used is the priority of attackers. TCP keeps software data synchronization reliable because it uses flow control and congestion control and keeps packets arrival in order. To reduce send times, most TCP programs use Nagle's algorithm, which accumulates data until it reaches Maximum Segment Size (MSS) size (1460 bytes) instead of sending small data immediately and frequently. This advantage in general software just needs to be disabled for game synchronization that is pursuing real-time. In the contrast, UDP provides unreliable connections for software data synchronization. Hence, some programs implement reliable UDP for data synchronization. In addition, TCP hole punching or Simple Traversal of UDP through NAT (STUN) is used to access computers inside Network Address Translation (NAT). Some software platforms only allow clients to send packets by IDs instead of IP addresses. However, the platform server IPs can be obtained by WireShark or Netstat.

Network latencies cannot be ignored even in the 5G era. For most software, network latencies include transmission delays, propagation delays in medium, queuing delays, and processing delays in routers. Either an unreliable physical layer, data link layer, or network layer may cause packet losses especially when the channel is full. In addition, a jitter may cause disordered arrival packets, which may lead to software stalls.

For 3D video games or other virtual reality software, input sampling latencies, render pipeline latencies and VSync latencies enlarge the distance to "current time", while players think they are interacting with others in real-time. Some online games use interpolations or predictions to fill in the time blanks. However, what if player_A is shooting at player_B in a prediction position when the prediction is wrong? Human hero Neo should be kept unaware he is in the Matrix. Some games use server-side replays to keep shooting without a miss if a player feels his shooting is accurate. What a coincidence, this mechanism can help reverse data modification for automatic aiming.

SMEO increases the execution time of protected software by dynamic obfuscation and self-monitor. Target codes are core codes that need to be hidden or protected. How to choose target codes is another technique, which is the weakness and limitation of the model, because it depends on the experiences of developers and experts.

5 Conclusions

The technology and trends of information security in different communication schemes, such as information encryption and data hiding are discussed in this paper. We presented implementation details in high-level programming languages (C++) so that readers can understand the proposed methodology for distributed software protections. In addition, hidden dangers of distributed software (especially games) are revealed. Detailed countermeasures for reverse data modifications are illustrated: Our overall idea to avoid reverse data modifications is that reversely do the reverse things. A novel obfuscation approach (SMEO) is proposed. The approach can periodically and automatically modify the protected software binary codes, and the protected software binary codes are always encrypted to resist static analysis and dynamic analysis. Furthermore, a simplified virtual machine is implemented to make the protected codes unreadable. Especially, SMEO is efficient to protect online computer 3D games. Experiments show that hidden dangers can be eliminated with our efficient methods.

Target codes are core codes that need to be hidden or protected. How to choose target codes is another technique. SMEO increases the execution time of protected software by dynamic obfuscation and self-monitor. Increment in the execution time of the target codes sometimes affects game performance but does not decrease players' game experiences. How do the actions of game players affect game performances in scenarios with and without protections? Different player actions and different protected code blocks affect the computation performances, which will be comprehensively analyzed in future work for different kinds (Massively Multiplayer Online, Simulations, Adventure, Real-Time Strategy, First Person Shooters, Sports, and Role-Playing) of computer games to help game developers validate performances in various online games.

Acknowledgement: The authors thank many researchers for their contributions and ideas.

Funding Statement: This work was supported by grants from Natural Science Foundation of Inner Mongolia Autonomous Region (No. 2022MS06024), NSFC (No. 61962040), Hainan Province Key R&D Program (ZDYF2022GXJS007, ZDYF2022GXJS010), Hainan Natural Science Foundation (620RC561), Hainan Province Higher Education and Teaching Reform Research Project (Hnjg2021ZD-3) and Hainan Province Key Laboratory of Meteorological Disaster Prevention and Mitigation in the South China Sea, Open Fund Project (SCSF202210).

Author Contributions: Conceptualization by Lei Yu and Yucong Duan; Methodology by Lei Yu; Design by Lei Yu; All authors read and approved the final manuscript.

Availability of Data and Materials: None.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antononglou, A. Huang *et al.,* "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 3, pp. 354–359, 2017.

[2] L. Yu and Z. Huo, "Reversely discovering and modifying properties based on active deep Q-learning," *IEEE Access*, vol. 8, pp. 157819–157829, 2020.

[3] C. Foket, K. D. Bosschere and B. D. Sutter. "Effective and efficient java-type obfuscation," *IEEE Software*, vol. 50, no. 2, pp. 136–160, 2020.

[4] M. R. Albrecht, P. Farshim, S. Han, D. Hofheinz, E. Larraiz *et al.,* "Multilinear maps from obfuscation," *Journal of Cryptology*, vol. 33, no. 3, pp. 1080–1113, 2020.

[5] J. Lai, Z. Huang, M. H. Au and X. Mao, "Constant-size CCA-secure multi-hop unidirectional proxy re-encryption from indistinguishability obfuscation," *Theoretical Computer Science*, vol. 847, pp. 1–16, 2020.

[6] D. Dachman-Soled, S. D. Gordon, F. H. Liu, A. O'Neill and H. S. Zhou, "Leakage resilience from program obfuscation," *Journal of Cryptology*, vol. 32, no. 3, pp. 742–824, 2019.

[7] R. Fellin and M. Ceccato, "Experimental assessment of XOR-masking data obfuscation based on K-clique opaque constants," *Journal of Systems and Software*, vol. 162, pp. 110492, 2019.

[8] J. Yang, X. Wang, Q. Zhou, Z. Wang, H. Li *et al.,* "Exploiting spin-orbit torque devices as reconfigurable logic for circuit obfuscation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 57–69, 2018.

[9] A. Sengupta, D. Kachave and D. Roy, "Low cost functional obfuscation of reusable IP cores used in CE hardware through robust locking," *IEEE Transactions on Computer-Aided Design Conference of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 604–616, 2018.

[10] X. Zhang and Y. Lao, "On the construction of composite finite fields for hardware obfuscation," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1353–1364, 2019.

[11] X. Wang, Q. Zhou, Y. Cai and G. Qu, "Toward a formal and quantitative evaluation framework for circuit obfuscation methods," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1844–1857, 2019.

[12] B. Olney and R. Karam, "Tunable FPGA bitstream obfuscation with boolean satisfiability attack countermeasure," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 2, pp. 1–22, 2020.

[13] L. Wang, D. Yang, X. Han, D. Zhang and X. Ma, "Mobile crowdsourcing task allocation with differential-and-distortion geo-obfuscation," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 967–981, 2019.

[14] T. Hou, Z. Qu, T. Wang, Z. Lu and Y. Liu, "ProTO: Proactive topology obfuscation against adversarial network topology inference," in *IEEE Conf. on Computer Communications*, Toronto, ON, CA, pp. 1598–1607, 2020.

[15] P. Corcoran, P. Mooney and A. Gagarin, "A distributed location obfuscation method for online route planning," *Computers & Security*, vol. 95, pp. 101850, 2020.

[16] S. Cao, G. Zhang, P. Liu, X. Zhang and F. Neri, "Cloud-assisted secure eHealth systems for tamper-proofing EHR via blockchain," *Information Sciences*, vol. 485, pp. 427–440, 2019.

[17] A. Sutton and R. Samavi, "Tamper-proof privacy auditing for artificial intelligence systems," in *Int. Joint Conf. on Artificial Intelligence Organization. JCAI*, Stockholm, AB, SE, pp. 5374–5378, 2018.

[18] A. C. Mert, E. Ozturk and E. Savas, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2019.

[19] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang *et al.,* "Semantics-aware obfuscation scheme prediction for binary," *Computers & Security*, vol. 99, pp. 102072, 2020.

[20] T. Bui, D. Cooper, J. Collomosse, M. Bell, A. Greeb *et al.,* "Tamper-proofing video with hierarchical attention autoencoder hashing on blockchain," *IEEE Transactions on Multimedia*, vol. 22, no. 11, pp. 2858–2872, 2020.

[21] M. Shayan, S. Bhattacharjee, A. Orozaliev, Y. Song, K. Chakrabarty *et al.,* "Thwarting bio-IP theft through dummy-valve-based obfuscation," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2076–2089, 2021.

[22] H. Zhou, T. Chen, H. Wang, L. Yu, X. Luo *et al.,* "UI obfuscation and Its effects on automated UI analysis for android apps," in *35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Virtual Event, AUS, pp. 199–210, 2020.

[23] X. Y. Wang, X. Y. Wang, B. Ma, Q. Li and Y. Q. Shi, "High precision error prediction algorithm based on ridge regression predictor for reversible data hiding," *IEEE Signal Process Letters*, vol. 28, pp. 1125–1129, 2021.

[24] B. Ma and Y. Q. Shi, "A reversible data hiding scheme based on code division multiplexing," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 1914–1927, 2016.