Overcoming Trial Version Software Cracking Using a Hybridized Self-Modifying Technique

C. K. Oputeh
Department of Computer Science
University of Port Harcourt,
Port Harcourt, Nigeria.

E. E. Ogheneovo
Department of Computer Science
University of Port Harcourt,
Port Harcourt, Nigeria.

Abstract
Information exchange has become an essential component in modern society. Vendors provide content to consumers, while consumers exchange information using e-mail, peer-to-peer systems, social networks, or other network applications. We rely on embedded software in our cars, the domotics, built into our homes, and other electronic devices on a daily basis. Obviously, all these applications rely on the correct functioning of software and hardware components. Often, software which is the driving force of computer hardware are usually subjected to cracking, a condition whereby hackers bypassing the registration and payments options on a software product to remove copyright protection safeguards or to turn a demo version of software into a fully functioning version by manipulating information such as the serial number, hardware key, dates, etc., without actually paying for the software. In this paper, we proposed a hybridized self-modifying technique for checking against cracking. Our technique combines obfuscation and hashing mechanisms to resist attackers from cracking software. The key idea is to hide the code using hashing by transforming it such that it becomes more difficult to understand the original source code and using obfuscation to resist software reverse engineering. The result shows that our technique is able to block hackers and thus prevent code cracking.

Keywords:
Software cracking, reverse engineering, code obfuscation, self-modification, encryption.

1.0 Introduction

Information exchange has become an essential component in modern society. Vendors provide content to consumers, while consumers exchange information using e-mail, peer-to-peer systems, social networks, or other network applications. We rely on embedded software in our cars, we trust the domotics (Home Automation) built into our homes, and we use electronic devices on a daily basis. Hence, the usage of software applications has become one of the corner stone of our lives. Obviously, all these applications rely on the correct functioning of software and hardware components [6]. According to Howard and LeBlanc [13], in the 1980s, application security was achieved through secure hardware, such as ATM terminals or set-top boxes. Since the 1990s, however, software protection has gained much interest due to its low cost and flexibility. Nowadays, we are surrounded by software applications, e.g., for online payments, social networking, games, etc. As a result, threats such as piracy, reverse engineering, and tampering have emerged. These threats are exacerbated by poorly protected software [5-7]. Therefore, it is important to have a thorough threat analysis as well as software protection schemes.

Today, the revenues of software companies are huge. Not only operating systems, but also professional applications (e.g. graphics software) can be very expensive. As a consequence, illegal use of software emerged [10] [17]. With just a few mouse clicks, people can download software; apply a downloaded patch to it, and start using it without payment. Vendors realized that protecting software against malicious users is a hard problem [9]. The user is in control of his machine: he has physical access to the hardware; he controls the network connectivity, etc. Nevertheless, software owners also manage to arm themselves against these threats. Examples include popular applications such as Apple’s media player - iTunes, the voice-over-IP application - Skype, or online games such as World of Warcraft [11]. These applications have been exposed to attacks over the years. Nevertheless, they still withstand the major problems caused by software threats such as reverse engineering, tampering, cracking or piracy [6] [8].

Software cracking is on the rampant due to the increase use of the Internet technology. Software cracking is the process of bypassing the registration and payments options on a software product to remove copyright protection safeguards or to turn a demo version of software into a fully function version without paying for it [3]. It involves the modification of software to remove or disable features which are considered undesirable by the person cracking the software, usually related to protection methods: copy protection (protection against the manipulation of software), trial/demo version, serial number, hardware key, date checks, CD check or software annoyances like nag screens and adware [9] [14]. Software cracking is a serious problem that and it possess a great danger to computer

Manuscript received April 5, 2015
Manuscript revised April 20, 2015

security. Therefore, there is need to provide protection to software in order to reduce in incidence of cracking.

In this paper, we study software cracking in relation to self-modification of programs. We proposed a hybridized self-modifying technique for overcoming trial version software cracking. First, we develop a framework for preventing code cracking, then we develop a model that can disguise the nature of code using self-modification. In section 2, we discuss related work. Section 3 discusses the methodology used. In section 4, we introduce our findings and the discussion of results. Finally, section 5 draws conclusion.

2.0 Related Work

Collberg and Thomborson [5] proposed software watermarking technique for preventing cracking. Watermarking is a compact outline of the approaches to protect against these threats. Software watermarking for instance focuses on protecting software reactively against piracy. It usually implants hidden, distinctive data into an application in such a way that it can be guaranteed that a particular software instance belongs to a particular individual or company. When this data is distinctive for each example, one can mark out copied software to the source unless the watermark is smashed. The second group, code obfuscation, protects the software from reverse engineering attacks. This approach comprises of one or more program alterations that alter a program in such a way that its functionality remains identical but analyzing the internals of the program becomes very tough. A third group of approaches focuses on making software “tamper-proof”, also called tamper-resistant.

Protecting the reliability of software platforms, particularly in unmanaged customer computing systems is a tough task. Attackers may try to carry out buffer overflow attacks to look for the right of entry to systems (access to system), steal secrets and patch on the available binaries to hide detection. Every binary has intrinsic weakness that attackers may make use of at any point in time. Srinivasan et al. [18] proposed three orthogonal techniques; each of which offers a level of guarantee against malware attacks beyond virus detectors. The techniques can be incorporated on top of normal defenses and can be integrated for tailoring the level of desired protection. The author tries to identify alternating solutions to the issue of malware resistance. The techniques used involve adding diversity or randomization to data address spaces, hiding significant data to avoid data theft and the utilization of distant evidence to detect tampering with executable code.

Protecting code against tampering can be regarded as the issue of data authenticity, where ‘data’ refers to the program code. Aucsmith [1] explained an approach to implement tamper resistant software. The approach protects against analysis and tampering. The author utilizes small, armored code segments, also called Integrity Verification Kernels (IVKs), to validate code integrity. These IVKs are protected via encryption and digital signatures in such a way that it is tough to modify them. Moreover, these IVKs can communicate with each other and across applications via an integrity verification protocol. Chang et al. [3] proposed an approach that depends on software guards. This protection technique is based on a composite network of software guards which mutually validate each other's consistency and that of the program's critical sections. A software guard is a small segment of code carrying out particular task, e.g. check summing or repairing. When check summing code discovers a modification; repair code is capable to undo this malevolent tamper challenge. The security of the approach depends partly on hiding the obfuscated guard code and the complexity of the guard network.

Horne et al. [12] using the same ideas as Chang et al. [3] proposed `testers', small hashing functions that validate the program at runtime. These testers can be integrated with embedded software watermarks to result in a unique, watermarked, self checking program. Other related research is unconscious hashing [4], which interweaves hashing instructions with program instructions and which is capable of proving whether a program is operated correctly. The approach used stochastic maintenance approach for software protection through the closed queuing system with the untrustworthy backups. The technique shows the theoretical software protection approach in the security viewpoint. If software application modules are denoted as backups under proposed structural design, the system can be overcome through the stochastic maintenance model with chief untrustworthy and random auxiliary spare resources with replacement strategies. Recently, Ge et al. [8] presented a research work on control flow based obfuscation. Although the authors contributed to obfuscation, the control flow data is protected with an Aucsmith-like tamper resistance approach.

Cappaert et al. [2] presented a partial encryption approach depending on a code encryption approach. In order to utilize the partial encryption approach, binary codes are partitioned into small segments and encrypted. The encrypted binary codes are decrypted at runtime by users. Thus, the partial encryption overcomes the faults of illuminating all of the binary code at once as only the essential segments of the code are decrypted at runtime. Jung et al. [16] presented a code block encryption approach to protect software using a key chain. Jung’s approach uses a unit block, that is, a fixed-size block, rather than a basic block, which is a variable-size block. Basic blocks refer to the segments of codes that are partitioned by control transformation operations, such as “jump” and “branch” commands, in assembly code [3] [15]. Jung’s approach is very similar to Cappaert’s scheme. Jung’s approach tries to solve the issue of Cappaert’s approach. If a block is invoked by more than two preceding blocks, the invoked block is duplicated.

3.0 Methodology

The software to be cracked must be a to be cracked must be a trail version that requires the use of serial number to unlock the software from a trail version software with limited features to a full version software with all the functionalities. Our sample software to be cracked is crackme.cpp. It is C++ program that displays the DOS environment screen that requires a serial key to unlock the software. At compilation time, an executable file version of the C++ source code is generated known as crackme.exe. If the serial key entered is valid, it displays correct key but if the key is invalid, it displays wrong key. The valid serial number 123 is used. Any other serial number entered apart from 123 will prompt the text string “wrong key”. The objective is to crack the software so that any key entered as serial number will unlock the software and display correct key.

3.1 Materials

We used four major software in this paper. They are:

- Code::Blocks 13.12
- MinGW Installer
- Hacker Disassembler (HDasm)
- Hex Editor (Hacker’s View-Hiew)

Code::Blocks is a free and open source, cross-platform IDE which supports multiple compilers including GCC and Visual C++. It is developed in C++ using wxWidgets as the GUI toolkits. Using a plugin architecture, its capabilities and features are defined by the provided plugins. Currently, Code::Blocks is oriented towards C, C++, and Fortran. Code::Blocks is being developed for Windows, Linux, and Mac OS X and has been ported to FreeBSD, OpenBSD, and Solaris. We used the Code::Blocks C++ compiler to compile the source code.

MinGW Installer formerly mingw32, is a free and open source development environment for native Microsoft Windows applications. It includes a port of the GNU Compiler Collection (GCC), GNU Binutils for Windows (assembler, linker, archive manager), a set of freely distributed Windows specific header files and static import libraries which enable the use of the Windows API, a Windows native built of the GNU debugger, and other utilities. The MinGW installer has all the runtime libraries needed by the Code::Blocks 13.12 in order to have access to all the runtime libraries available during compilation of the source code.

Hacker’s Disassembler (HDasm) is a disassembler, for computer software which automatically generates assembly language source code from machine-executable code. It supports a variety of executable formats for different processors and operating systems. HDasm performs automatic code analysis, using cross-references between code sections, knowledge of parameters of API calls, and other information.

Hex Editor (or binary file editor or byte editor) is a type of computer program that allows for manipulation of the fundamental binary data that constitutes a computer file. The name ‘hex’ comes from ‘hexadecimal’: the standard numerical format for editing binary data. It contains hacker’s view (Hiew) which has the ability to view files in text, hex, and disassembly modes. The program is particularly useful for editing executable files.

3.2 Method

Our model is a hybridized method to protect software against illegal acts of hacking. We examine software protection through code obfuscation and encryption technique, known as one - way hashing, which resists reverse engineering attacks. Our model is a code transformation technique in which functionality of original code is maintained while obfuscated and one - way hashed code is made difficult to reverse engineer. The key idea is to hide the code. In our technique, the application is transformed so that it is functionally identical to the original but it is much more difficult to understand. This is done by adding a mechanism of self-modification, known as obfuscation mechanism, to the original program, so that it becomes hard to be analyzed. In the binary program obtained by the proposed method, the original code fragments we want to protect are obfuscated so that the hackers would not be able to understand the real source code. Then, we use an encryption technique, known as one - way hashing, to generate our application licenses.

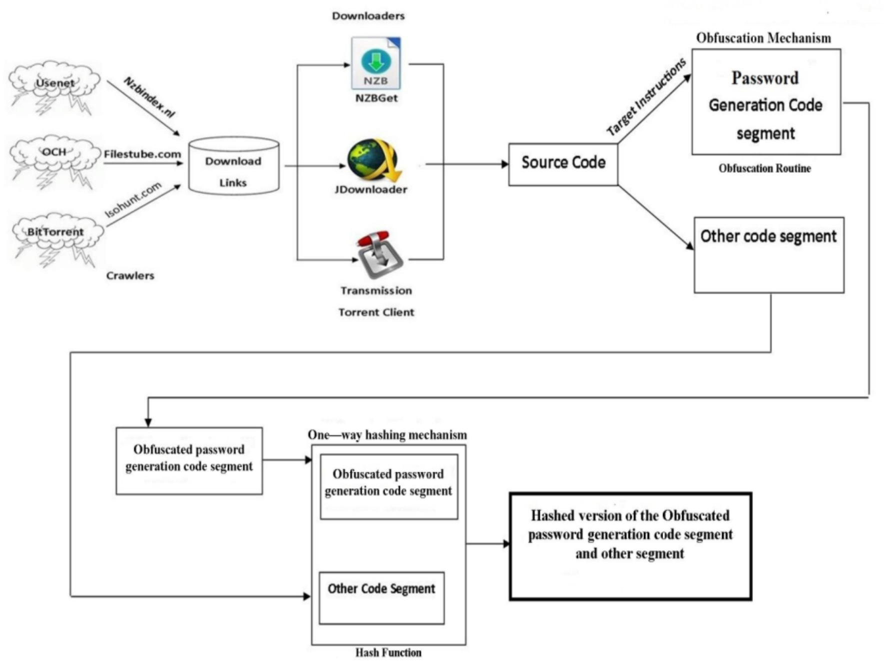

Downloaders
Usenet
Nzbindex.nl
OCH
Filestube.com
Download Links
Isohunt.com
BitTorrent
Crawlers
NZB
NZBGet
JDownloader
Transmission Torrent Client
Source Code
Target Instructions
Obfuscation Mechanism
Password Generation Code segment
Obfuscation Routine
Other code segment
Obfuscated password generation code segment
One—way hashing mechanism
Obfuscated password generation code segment
Other Code Segment
Hash Function
Hashed version of the Obfuscated password generation code segment and other segment

Fig. 1: Architecture of our Hybridized Self-modifying Mechanism

Our approach primarily consists of two parts: First, we hide our target instructions. Target instructions are the essential parts of the source code that we intend hiding. In this technique, the target instructions are the code segment of the serial number/password generation scheme. We introduce an encryption mechanism, known as one - way hashing, to hide the password of the application within the program code. This is done by hashing the password to generate a hashed version of the password. If a hacker obtains access to the password file, all he or she would see would be a collection of mashed data. Secondly, to ensure that reverse engineering or analysis is not performed on the source code, we add obfuscating instructions that obfuscates both the hashed password and the remaining segments of the source code.

Thus, by using the hybridized self-modification mechanism, we used one - way hash the password to hide the password and obfuscate the original source code. A cracker with the intention of cracking software would not be able to crack it because the sight of the obfuscated source code does not look like a conventional written program. We believe that the program protected by our method is quite hard to be understood, and that it is difficult for crackers to cancel the protection, since the program is both obfuscated and one - way hashed.

3.2.1 Obfuscation Mechanism

Our obfuscation mechanism utilizes an algorithm for the obfuscation of the serial number generation code segment and the other code segment. The algorithm is written below:

Algorithm 1: Obfuscation Mechanism

```
1.  Init sl = strings[]
2.  Init ia = address of sl
3.  Init al = argument list of recursive function
4.  FOR x = 1 to length of al
5.      al = al + sl[x]
6.  END FOR
7.  Insert 3 lower case l's into a Boolean statement
8.  FOR x = 1 to length of al
9.          IF x is at the proper position of sl, then
10              print 'l'
11.         END IF
12.      Init al[x+1] = printed "l"
13.      SUBTRACT n from x to get element of sl
14.      SWAP conditional operators
15.    WHILE (Swapping between integers and characters) DO
16.         RENAME variables
17.    END WHILE
18. END FOR
19. RENAME functions to look like variable names
20. ELIMINATE argument type specifiers
```

3.2.2 One - Way Hashing Mechanism

In the one - way hashing mechanism module, we are interested in the password generation code segment. This is because this code segment is responsible for generating the serial number/password of the software. Thus we must hide or conceal it so it will not be visible for the hacker or cracker to see. When this code segment is one - way hashed, the cracker will not be able to see the password, rather, what will be visible is be mashed data. This implies that the cracker may not even know that the software requires a serial number to have access to the full functionality of the software. We only subject the serial number generation code segment to one - way hashing mechanism while the other code segments are neither encrypted nor one - way hashed. Figure 2 shows a one-way hashing mechanism also referred to as encryption. First, the obfuscated password is generated from the code segment and secondly, the other code segment is also produced. Both processes are then used in the hashing function to encrypt the code and prevent it from being understood by a cracker who wants to crack the code and use it for personal purpose.

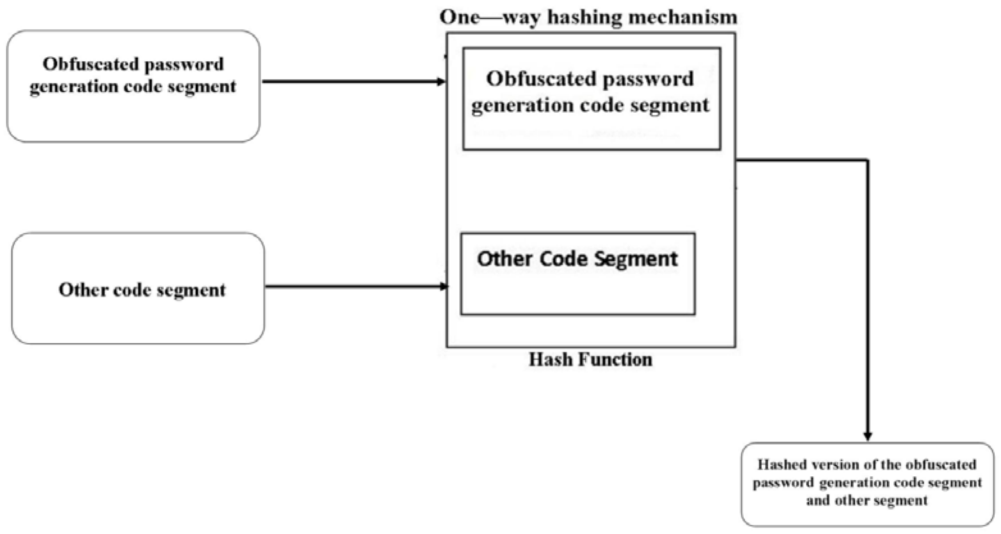

One—way hashing mechanism
Obfuscated password generation code segment
Obfuscated password generation code segment
Other code segment
Other Code Segment
Hash Function
Hashed version of the obfuscated password generation code segment and other segment

Fig. 2: Encryption (One - Way Hashing) Mechanism Module

The one - way hashing mechanism utilizes an algorithm (algorithm 1) for the hashing of the serial number generation code segment and the other code segment. The algorithm is written below:

Algorithm 2: One - way hashing Mechanism

```
 1:  DECLARE hash, file_size
 2:  Open file for reading and writing in binary mode
 3:  Init file pointer to beginning of file
 4:  Init f = file current read position
 5:  Reset file pointer to beginning of file
 6:  Init hash = f
 7:  Init i = 0, tmp = 0,
 8:  FOR j = 1 to 10
 9:     WHILE 65536/(size of tmp) DO
10:           Read characters from file
11:           Store characters in tmp location
12:           WHILE (not eof) DO
13:           Read characters from file
14:        END WHILE
15:     END WHILE
16:        Init file pointer to 65536 - 1
17:        Init hash = tmp + hash
18:        increment i
19:  END FOR            // end the for loop
20:  RETURN hash
```

Our hybridized self - modifying mechanism is designed using UML (Unified Modeling Language) diagram. An activity diagram of our mechanism is shown in figure 2.

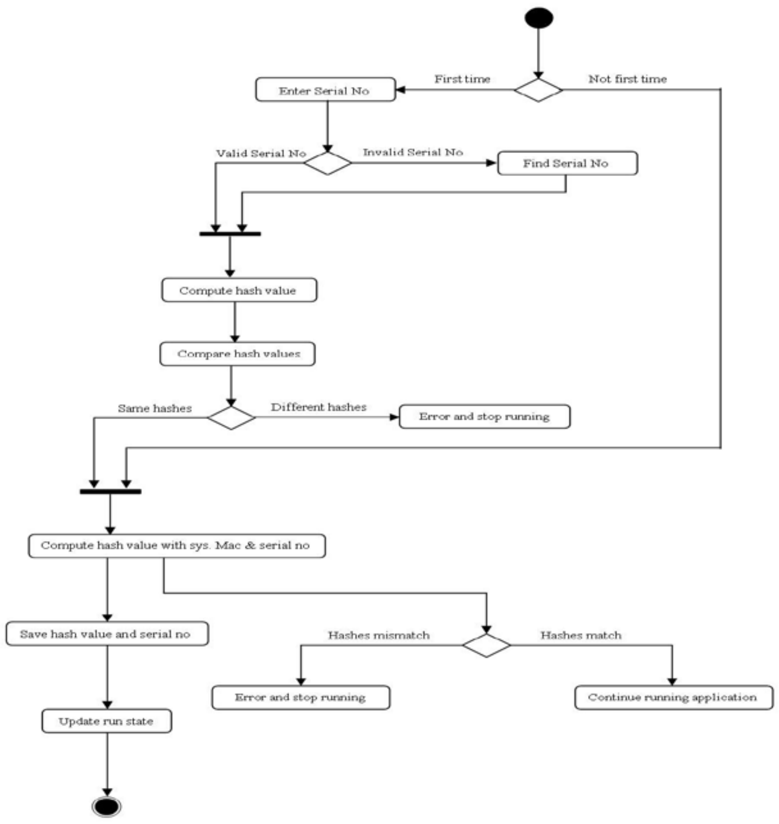


Fig. 3: An activity diagram showing the hybridized self - modifying mechanism

Figure 3 shows the activity diagram for modeling the hybridized self-modifying technique used employed. There are two files needed to run this application software. The application executable file and the configuration file. When you run the application (the executable file) for the first time, you are prompted to enter the serial number. Then the serial number is entered in the application interface. The application then checks if the serial number is valid. If the serial number is invalid, then the application stops running. If the serial number is valid, an hash value is computed and compared with the existing hash value in the configuration file. If the hash values are different, application stops running. If the hash values are the same, then a new hash value is computed using the system mac address and the serial number. These new computed hash value and serial number are then stored in the configuration file and an update of the present run - state of the application software is stored in the configuration file.

4.0 Results and Discussions

We used sample software to be cracked called crackme.cpp. It is C++ program that displays the DOS environment screen that requires a serial key to unlock the software. At compilation time, an executable file version of the C++ source code is generated known as crackme.exe. If the serial key entered is valid, it displays correct key but if the key is invalid, it displays wrong key. The valid serial number 123 is used. Any other serial number entered apart from 123 will prompt the text string “wrong key”. The objective is to crack the software so that any key entered as serial number will unlock the software and display correct key. Figure 4 shows a Hiew (Hexadcimal view) displaying crackme.exe in text format.

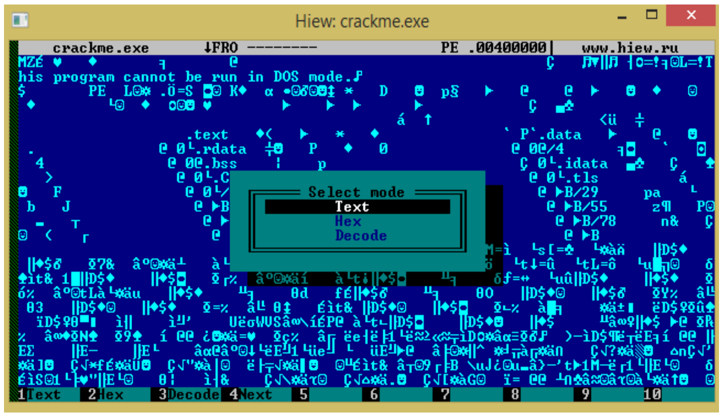
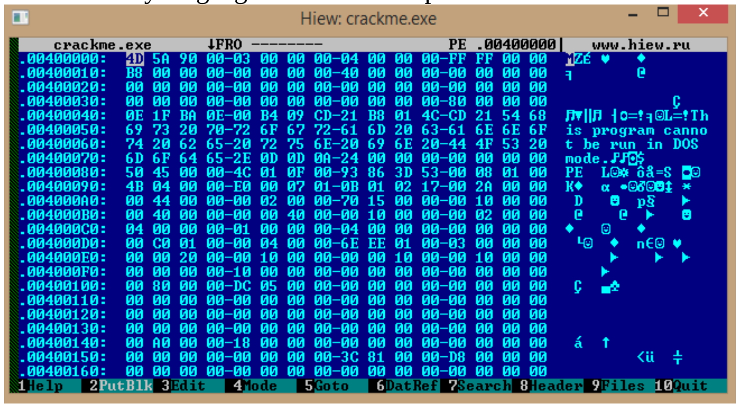

Hiew: crackme.exe
crackme.exe ↓FRO -------- PE .00400000 www.hiew.ru
This program cannot be run in DOS mode.
Select mode
Text
Hex
Decode
1Text 2Hex 3Decode 4Text 5 6 7 8 9 10

Fig. 4: Hiew displaying crackme.exe in text format

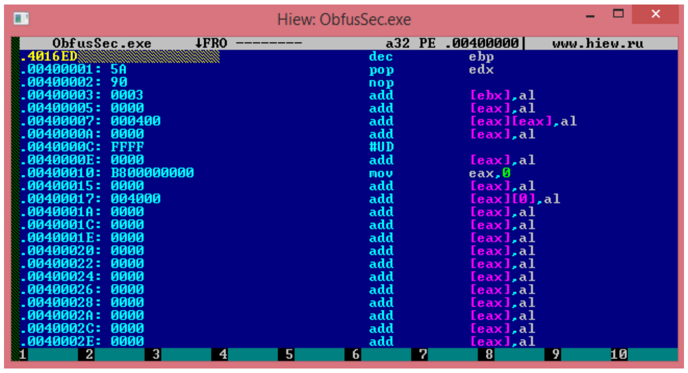
Figure 5 shows Hiew displaying crackme.exe in hexadecimal format which is the assembly language format representing the both the OPCODE, the hexadecimal representation and the text formats.



```
Hiew: crackme.exe
   crackme.exe        ↓FRO --------              PE .00400000│    www.hiew.ru
.00400000:  4D 5A 90 00-03 00 00 00-04 00 00 00-FF FF 00 00  MZÉ
.00400010:  B8 00 00 00-00 00 00 00-40 00 00 00-00 00 00 00
.00400020:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.00400030:  00 00 00 00-00 00 00 00-00 00 00 00-80 00 00 00
.00400040:  0E 1F BA 0E-00 B4 09 CD-21 B8 01 4C-CD 21 54 68  Th
.00400050:  69 73 20 70-72 6F 67 72-61 6D 20 63-61 6E 6E 6F  is program canno
.00400060:  74 20 62 65-20 72 75 6E-20 69 6E 20-44 4F 53 20  t be run in DOS
.00400070:  6D 6F 64 65-2E 0D 0D 0A-24 00 00 00-00 00 00 00  mode.
.00400080:  50 45 00 00-4C 01 0F 00-93 86 3D 53-00 08 01 00  PE
.00400090:  4B 04 00 00-E0 00 07 01-0B 01 02 17-00 2A 00 00
.004000A0:  00 44 00 00-00 02 00 00-70 15 00 00-00 10 00 00
.004000B0:  00 40 00 00-00 00 40 00-00 10 00 00-00 02 00 00
.004000C0:  04 00 00 00-01 00 00 00-04 00 00 00-00 00 00 00
.004000D0:  00 C0 01 00-00 04 00 00-6E EE 01 00-03 00 00 00
.004000E0:  00 00 20 00-00 10 00 00-00 00 10 00-00 10 00 00
.004000F0:  00 00 00 00-10 00 00 00-00 00 00 00-00 00 00 00
.00400100:  00 80 00 00-DC 05 00 00-00 00 00 00-00 00 00 00
.00400110:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.00400120:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.00400130:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.00400140:  00 A0 00 00-18 00 00 00-00 00 00 00-00 00 00 00
.00400150:  00 00 00 00-00 00 00 00-3C 81 00 00-D8 00 00 00
.00400160:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
1Help  2PutBlk 3Edit  4Mode  5Goto  6DatRef 7Search 8Header 9Files 10Quit
```

Fig. 5: Hiew displaying crackme.exe in Hexadecimal format

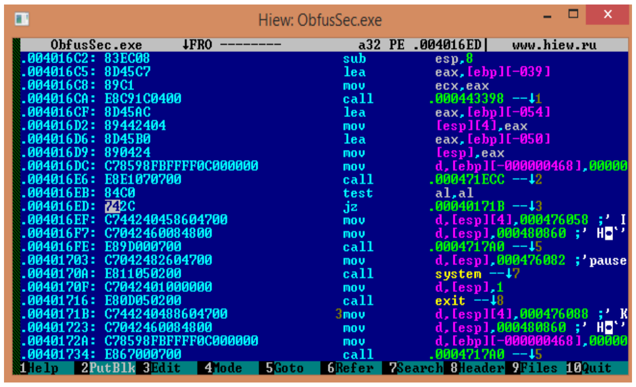
Using the same cracking rules as crackme.exe software that has a serial key authentication attribute, we generated some program outputs to show the result as we undergo the cracking process. We compiled the self-modifying code and the executable named ObfusSec.exe is then generated at compile time. We then run the source code to check the serial number parameters. If we enter the serial number "123" it will indicate that the key is valid and we can then crack the code. However, if we enter any other key, it will indicate "invalid" showing that that is not the correct serial number (key). Figure 6 shows the assembly language representation of the obfusSec.exe trying to search for offset address. This is the part we are more interested in because it shows that the source code can be generated in assembly language and it show the offset addresses.



```
Hiew: ObfusSec.exe
   ObfusSec.exe       ↓FRO --------          a32 PE .00400000│    www.hiew.ru
.4016ED                                   dec         ebp
.00400001: 5A                             pop         edx
.00400002: 90                             nop
.00400003: 0003                           add         [ebx],al
.00400005: 0000                           add         [eax],al
.00400007: 000400                         add         [eax][eax],al
.0040000A: 0000                           add         [eax],al
.0040000C: FFFF                           #UD
.0040000E: 0000                           add         [eax],al
.00400010: B800000000                     mov         eax,0
.00400015: 0000                           add         [eax],al
.00400017: 004000                         add         [eax][0],al
.0040001A: 0000                           add         [eax],al
.0040001C: 0000                           add         [eax],al
.0040001E: 0000                           add         [eax],al
.00400020: 0000                           add         [eax],al
.00400022: 0000                           add         [eax],al
.00400024: 0000                           add         [eax],al
.00400026: 0000                           add         [eax],al
.00400028: 0000                           add         [eax],al
.0040002A: 0000                           add         [eax],al
.0040002C: 0000                           add         [eax],al
.0040002E: 0000                           add         [eax],al
1      2      3      4      5      6      7      8      9      10
```

Fig. 6: Assembly language format of obfusSec.exe searching for offset address

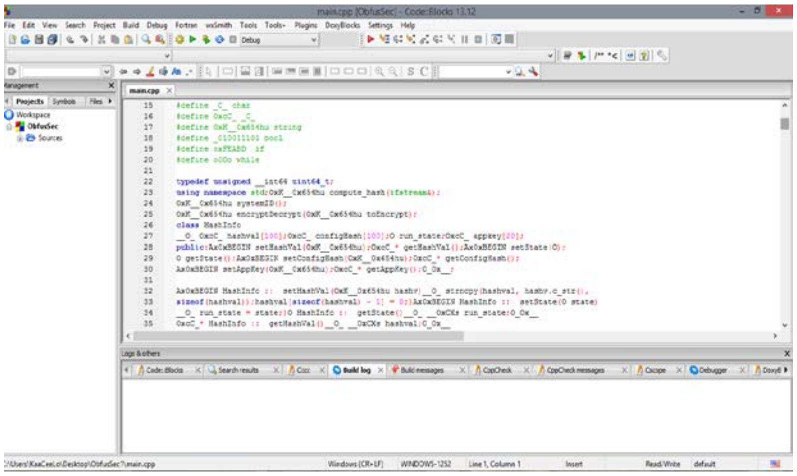
The program was later run as an update of the one in figure 6 when we search for the offset address. This is shown in figure 7.



```
 ObfusSec.exe      ↓FRO --------              a32 PE .004016ED|    www.hiew.ru
.004016C2: 83EC08                            sub       esp,8
.004016C5: 8D45C7                            lea       eax,[ebp][-039]
.004016C8: 89C1                              mov       ecx,eax
.004016CA: E8C91C0400                        call     .000443398 --↓1
.004016CF: 8D45AC                            lea       eax,[ebp][-054]
.004016D2: 89442404                          mov       [esp][4],eax
.004016D6: 8D45B0                            lea       eax,[ebp][-050]
.004016D9: 890424                            mov       [esp],eax
.004016DC: C78598FBFFFF0C000000              mov       d,[ebp][-000000468],00000
.004016E6: E8E1070700                        call     .000471ECC --↓2
.004016EB: 84C0                              test      al,al
.004016ED: 742C                              jz       .00040171B --↓3
.004016EF: C744240458604700                  mov       d,[esp][4],000476058 ;' I
.004016F7: C7042460084800                    mov       d,[esp],000480860 ;' H▪`'
.004016FE: E89D000700                        call     .0004717A0 --↓5
.00401703: C7042482604700                    mov       d,[esp],000476082 ;'pause
.0040170A: E811050200                        call      system --↓7
.0040170F: C7042401000000                    mov       d,[esp],1
.00401716: E80D050200                        call      exit --↓8
.0040171B: C744240488604700                 3mov       d,[esp][4],000476088 ;' K
.00401723: C7042460084800                    mov       d,[esp],000480860 ;' H▪`'
.0040172A: C78598FBFFFF0C000000              mov       d,[ebp][-000000468],00000
.00401734: E867000700                        call     .0004717A0 --↓5
1Help  2PutBlk 3Edit  4Mode  5Goto  6Refer 7Search 8Header 9Files 10Quit
```

Fig. 7: Offset address spotted with the OPCODE and assembly command jz

We search for the offset address down initially and then type the offset address without the two zero at the beginning of the offset address and then edited the code. As shown in figure 7, we edited the OPCODE by changing the 74 to 75, then we observed that the assembly command changes from **jz** to **jnz**. We the run the program and then insert serial number of our choice. If any serial key entered grants us access, then the software has been cracked.



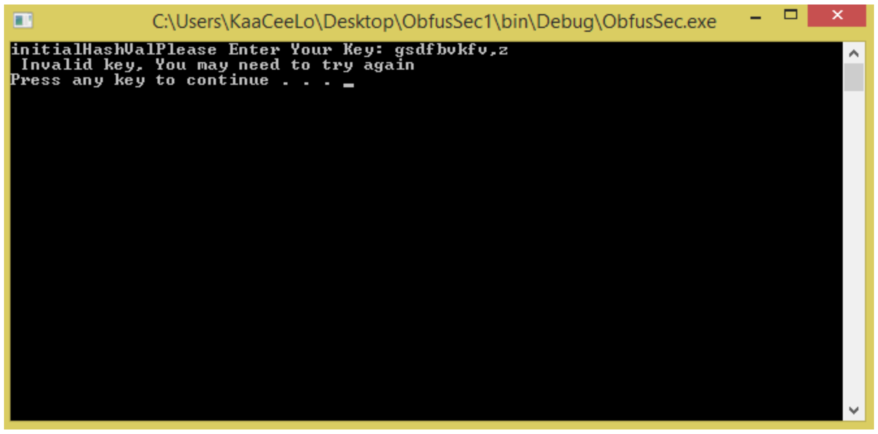
```
#define _C_ char
#define OxcC_ _C_
#define OxK__Cx654hu string
#define _010011101 bool
#define oxFEABD if
#define o0Oo while

typedef unsigned __int64 uint64_t;
using namespace std;OxK__Cx654hu compute_hash(ifstream&);
OxK__Cx654hu systemID();
OxK__Cx654hu encryptDecrypt(OxK__Cx654hu toEncrypt);
class HashInfo
__O_ OxcC_ hashval[100];OxcC_ configHash[100];O run_state;OxcC_ appkey[20];
public:AxOxBEGIN setHashVal(OxK__Cx654hu);OxcC_* getHashVal();AxOxBEGIN setState(O);
O getState();AxOxBEGIN setConfigHash(OxK__Ox654hu);OxcC_* getConfigHash();
AxOxBEGIN setAppKey(OxK__Cx654hu);OxcC_* getAppKey();C_Ox__;

AxOxBEGIN HashInfo ::  setHashVal(OxK__Ox654hu hashv)__O_ strncpy(hashval, hashv.c_str(),
sizeof(hashval));hashval[sizeof(hashval) - 1] = 0;}AxOxBEGIN HashInfo ::  setState(O state)
__O_ run_state = state;}O HashInfo ::  getState()__O_ __OxCXs run_state;C_Ox__
OxcC_* HashInfo ::  getHashVal()__O_ __OxCXs hashval;C_Ox__
```

Fig. 8: The source code of the obfuscated.exe

In Figure 8, the C++ source code of our model is opened using the Code::Blocks 13.12 IDE. This code shows that the source code has been obfuscated and cannot be understood by a software cracker even though the code remains the same as the original code written in C++.



```
initialHashValPlease Enter Your Key: gsdfbvkfv,z
 Invalid key, You may need to try again
Press any key to continue . . . _
```

Fig. 9: An invalid result when a wrong serial key is entered.

Figure 9 shows the DOS version where we entered a key different from the serial key. Since the result is “invalid”**,** it means that the source code remains uncracked after using all the normal cracking routine.

5.0 Conclusion

In this paper, we proposed a hybridized self-modifying technique for checking against cracking. Our technique combines obfuscation and code encryption techniques to resist attackers from cracking software. The key idea is to hide the code using obfuscation by transforming it such that it becomes more difficult to understand the original source code. Target instructions are the essential parts of the source code that we intend hiding. In this technique, the target instructions are the code segment of the serial number/password generation scheme. We introduce an encryption mechanism, known as one - way hashing, to hide the password of the application within the program code. This is done by hashing the password to generate a hashed version of the password. If a hacker obtains access to the password file, all he or she would see would be a collection of mashed data. Secondly, to ensure that reverse engineering or analysis is not performed on the source code, we add obfuscating instructions that obfuscates both the hashed password and the remaining segments of the source code. Thus, by using the hybridized self-modification mechanism, a cracker with the intention of cracking software would not be able to crack it because the sight of the obfuscated source code does not look like a conventional written program. We believe that the program protected by our method is quite hard to be understood, and that it is difficult for crackers to cancel the protection, since the program is both one - way hashed and obfuscated.

References

[1] A. Aucsmith. Tamper-Resistance Software: An Implementation. In Ross Anderson, Editor, Information Hiding, Proceedings of the First International Workshop, volume 1174 of LNCS, pp. 317 – 333.

[2] J. Cappaert, N. Kisserli, D. Schellekens and B. Preneel. Self-Encrypting Code to Protect Against Analysis and Tampering, 1st Benelux Workshop Inf. Syst. Security, 2006.

[3] H. Chang and M. Atallah. Protecting Software Codes by Guards, ACM Workshop on Digital Rights Management (DRM 2001), LNCS 2320, pp. 160-175, 2001.

[4] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha and M. Jakubowski. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive, Proc. 5th Information Hiding Workshop (IHW), Netherlands (October 2002), Springer LNCS 2578, pp. 440 – 414, 2002.

[5] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, And Obfuscation - Tools for Software Protection, IEEE Transactions on Software Engineering, Vol. 28, Issue: 8, pp. 735 – 746, 2002.

[6] S. Debray and J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In Antoniol G., Pinzger, M. and Chikofsky, E. J., Editors, WCRE, IEEE Computer Society, pp. 131-140, 2010.

[7] P. Djekic, and C. Loebbecke. Preventing Application Software Piracy: An Empirical Investigation of Technical Copy Protections. The Journal of Strategic Information Systems, Vol. 16, No. 2, pp. 173-186, 2007.

[8] J. Ge, S. Chaudhuri and A. Tyagi. Control Flow Based Obfuscation**.** In DRM '05: Proceedings of the 5th ACM workshop on Digital rights management, pp. 83-92, 2005.

[9] S. Goode and S. Cruise. What Motivates Software Crackers? Journal of Business Ethics, vol. 65, pp. 173-201, 2006.

[10] R. Gopal and G. Snaders. International Software Piracy: Analysis of Key Issues and Impacts. Info. Sys. Research, Vol. 9, No. 4, pp. 380-397, 1998.

[11] Y. Gu, B. Wyseur and B. Preneel. Software-Based Protection is Moving to the Mainstream, IEEE Software, Special Issue on Software Protection, Vol. 28, No. 2, pp. 56-59, 2011.

[12] B. Horne, L. Matheson, C. Sheehan and R. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.141–159, 2002.

[13] M. Howard and D. LeBlanc. Writing Secure Code, Second Edition. Microsoft Press, 2002

[14] A. Jain, K. Jason, S. Jordan and T. Brian (2007). Software Cracking (April 2007) **website**

[15] M. Jakobsson and M. Reiter. Discouraging Software Piracy Using Software Aging, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.1–12, 2002.

[16] D. Jung, H. Kim and J. Park. A Code Block Cipher Method to Protect Application Programs From Reverse Engineering, Korea Inst. Inf. Security Cryptology, Vol. 18, No. 2, pp. 85-96, 2008.

[17] M. Kammerstetter, C. Platzer and G. Wondracek. **Vanity, Cracks and Malware: Insights into the Anti-Copy Protection Ecosystem,** Proceedings of The 2012 ACM Conference On Computer And Communications Security, pp. 809-820, .2012

[18] R. Srinivasan, P. Dasgupta, V. Iyer, A. Kanitkar, S. Sanjeev and J. Lodhia. A Multi-factor Approach to Securing Software on Client Computing Platforms, 2010 IEEE Second International Conference on Social Computing (SocialCom), pp. 993 – 998, 2010.