All rights are reserved and copyright of this manuscript belongs to the authors. This manuscript have been printed and distributed without reviewing and editing as received from the authors: posting the manuscript to SCIS 2005 does not prevent future submission to any journals or conferences with proceedings.

**SCIS 2005** The 2005 Symposium on Cryptography and Information Security
Maiko Kobe, Japan, Jan.25-28, 2005
The Institute of Electronics, Information and Communication Engineers

# Tamper Resistant Software Through Dynamic Integrity Checking

Ping Wang * Seok-kyu Kang * Kwangjo Kim *

**Abstract**— Code modification is the main method for software piracy. Making software tamper resistant is the challenge for software protection. In this paper, we present and explore a methodology that we believe can protect program integrity in a more tamper-resilient and flexible manner. we describe a dynamic integrity verification mechanism designed to prevent modification of software. The mechanism makes use of multi-blocking encryption technique so that no hash value comparison is needed and if the program was altered, the program will not exit in a traceable way. We also make use of common virus techniques to enhance our security. Our mechanism operates on binaries that can be applied to all PE format files like EXE and DLL. The overhead in runtime execution and program size is reasonable as illustrated by real implementation. The scheme is built to be compatible with code obfuscation and other tamper resistance techniques.

**Keywords:** software piracy, tamper resistant, integrity checking, multi-blocking encryption.

## 1 Introduction

Development of technology for the prevention of software piracy is important for the software industry. The advent of mobile computing will only make the problem of software piracy worse. In the near future it will be commonplace for users to carry applications on a mobile computing device. However, these applications would often be transferred and executed on remote computing servers that would be part of the ubiquitous computing infrastructure. In light of the greatly improved computing power of current processors, it is acceptable to expand a fraction of this computing power on protecting software. The goal of this work is a naive approach to prevent malicious users of the software from creating fully functioning unauthorized copies of protected software.

Tamper resistance is the art and science of protecting software or hardware from unauthorized modification and distribution. Algorithms like MD5 and CRC are commonly used for integrity checking of the software. A common approach is to hash the whole block of software to obtain a hash value. To check the integrity of that software, this hash value will be compared with the hash value calculated based on the current copy of the software before running. If the two hash values do not match, the software has probably been modified and the program will be terminated. However, this static hash value checking is easily bypassed by locating the hash value comparison instruction and modifying the binary program code with existing software debugging tools. This branching instruction that performs the hash values comparison becomes a single point of failure.

In this paper, we initiate the use of multi-blocking encryption technique [2], which was originally used to resist code observation, in integrity checking and propose a multi-block hashing scheme. The mechanism behind multi-blocking encryption is to break up a binary program into individually encrypted segments. We base on this mechanism to perform the integrity checking. Roughly speaking, we will take the hash value of a block as the secret key for decrypting the next block. The advantages of our approach include the followings: No hash value checking is needed. If the program was altered, the hash value is changed and therefore the next block cannot be decrypted properly. Due to the corruption of the next block, the program cannot continue to run.

Also, using this dynamic multi-block hashing scheme, the integrity of the software is kept checking during the program run-time execution. The adversary (*i.e.*, the software pirate) is unable to obtain a single point of failure. Unlike the previous approaches that rely on branching instructions for checking the hash values, in our approach, we do not have any such instructions. The technique of bypassing the checking instructions is no longer feasible in our approach. On the other hand, if the program was altered, the program will not exit immediately. Therefore the adversary is very difficult to trace back the problem. The feasibility of our suggested approach is realized by a real implementation. Experimental results show that the overhead in runtime execution as well as the increase in program size is reasonable.

The rest of the paper is organized as follows: Section 2 reviews existing tamper proofing techniques. Section 3 discusses the idea of our multi-block hashing scheme and its security issues. Section 4 shows the experimental results. The last section discusses the generalization of our approach and the conclusion.

* International Research center for Information Security (IRIS), Information and Communications University (ICU), 119 Munji-ro, Yusong-gu, Daejeon, 305-714, Korea, Tel: +82-42- 866-6236, Fax: +82-42-866-6273, ({pwang, redorb, kkj}@icu.ac.kr)

## 2 Related Work

In the last few years, there have been active development of software obfuscation [7] and watermarking [5] but only a few researches have been done in software integrity verification. Computing a hash value of code bytes is a common integrity verification method. It examines the current copy of the executable program to see if it is identical to the original one by checking the hash values. The main drawback of this approach is that the adversary can easily bypass the verification by locating the hash value comparison instruction. Furthermore, since this method only verifies the static shape of the code, it cannot detect run-time attacks, where the debugger tools monitor the program execution and the adversary can identify the instructions that are being executed and then modify them.

Hashing functions scan a block of the program, and use the data contained therein as input to a mathematical equation. The simplest way of which would be to sum all the numbers in a given block. When done, the output must agree with the previously determined result. If they are not the same, this block of the program has been altered.

It is fairly obvious that such a scheme is easily broken, which has led to more complex techniques. Some techniques may use values on the stack at a crucial instant in time, or values in a register. Others perform complex mathematical equations on overlapping sections of code. Horne *et al*. [9] have implemented such a system, using linear hash functions, which overlap and also hash the hashing functions as well. One of the strong points of hashing is that you can have as many hashing functions as you want, all performing a different hash function.

In order to detect run-time attacks, Chen and Venkatesan proposed oblivious hashing [4] based on the actual execution of the code. This method examines the validity of intermediate results produced by the program. It is accomplish by injecting additional hashing codes into the software. These hash codes are calculated by taking the results of previous instructions from the memory. In other words, the hash values are calculated based on the dynamic shape (run-time states) of the program so as to make it more difficult to attack. However, there is a practical constraint for binary-level code injection and if the adversary can locate the instructions for hash value comparison, bypassing is still possible.

Chang and Atallah [3] proposed another method that enhances the run-time protection in which protection is provided by a network of execution unit called guard. A guard regarded as a small code segment performs checksums on part of the binary code to detect if the software has been modified. The guards are inserted into the software with different locations. They are inter-related so as to form a network of guards that reinforce the protection of one another by creating mutual protection.

They also proposed the use of guards to repair attacked code. Although a group of guards are more resilient against attacks than a single branching instruction for comparing hash values, it only spreads out the single attack point into different locations of the program. In other words, although more complicated, bypassing instructions performed by the guards is still possible.

The above integrity checking techniques all involve hash value comparison, which can be quite easily bypassed. Recently, Collberg and Thomborson [6] discussed an innovative idea that suggests encrypting the executable, thereby preventing anyone from modifying it successfully unless the adversary is able to decrypt it. However, the details are not discussed in their paper and only very little researches focusing on this kind of technology can be found. Our suggested approach can be regarded as the same direction as the idea proposed by them in which encryption is used instead of hash value comparison.

## 3 Multi-block Hashing Scheme

We propose the use of multi-blocking encryption for the integrity verification of software. In this section, we will first introduce the proposed multi-blocking encryption, then followed by the hashing scheme. Its security issues will also be discussed.

Multi-blocking encryption breaks up a binary program into individual encrypted blocks. The program is executed by decrypting and jumping to the executable block during the run-time process. When not being executed, blocks are in encrypted form after applying this program protection technique, therefore the adversary cannot modify the code statically, where the program being disassembled is examined by the disassembler which is not able to interpret the encrypted version of it. To our knowledge, Aucsmith [2] was the first to introduce the concept of multi-blocking encryption. The armored segment of code, which call integrity verification kernel (IVK), was used to construct tamper resistant software. The IVK is divided into several equal size cells, which are encrypted. Cells are exposed by decryption as it is executed. The multi-blocking encryption technique used in [2] is mainly used to resists code observation. In contrast to this approach, we make use of this technique to resist code modification during program execution.

### 3.1 Proposed Multi-blocking Encryption

Based on the concept of multi-blocking encryption in [2], we propose to apply this technique in the following way. We propose to divide a program into several different sized blocks (instead of equal sized blocks) according to the flow of the program. Each block is encrypted with a different key, which is illustrated in Figure 1. Let the program be P. If it can be broken down in several blocks, each basic block has the property of being independent. This means that the block does not have any jump/branch instructions, jumping/branching to other blocks. In other words, all the targets of the jump/branch instructions are local within the block. In order to find out the basic blocks, we first need to disassemble the executable program P to its

machine code instructions accurately. There are two generally used techniques for this: linear sweep and recursive traversal [10].

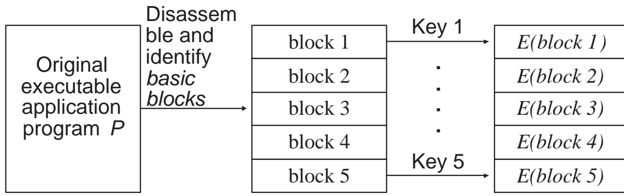


Figure 1: Different sized blocks encrypted by different keys

In general, it is convenient and feasible to store those keys inside the hardware token [11] so that dumping of keys from the main memory is impossible. The encryption can also be done inside the token. In our approach, we make use of the hash values of the blocks to be the encryption keys, thus further eliminate the necessity of storing the keys (see Section 3.3 for details).

### 3.2 Determining the Number of Blocks

The number of blocks ranges from the whole program (one-time encryption) to one instruction per block. It is clear to see that in the extreme environment, of which each block is one instruction, it can achieve the maximum-security level. The instruction can be decrypted inside a special CPU as well. However, it is infeasible to put such heavy workload into the CPU itself. Thus, the number of blocks should be determined by striking a balance between the level of security to be achieved and the speed of the program. Note also that the blocks also depend on the actual control flow of the program.

### 3.3 Multi-block Hashing Scheme

In this subsection, we will describe our proposed multi-block hashing scheme. Our objective is to prevent an adversary from modifying the software. Assuming that a user has legal access to the software, he may try to tamper it to remove authentication code so that it can be freely distributed for illegal use.

Our scheme works as follows: We take the hash value of a basic block as the secret session key for decrypting the next basic block according to the flow of the program. As an initiate study, we focus on the programs of which the control flow is a tree-like structure (as shown in Figure 2). We remark that this kind of control flow is very common.

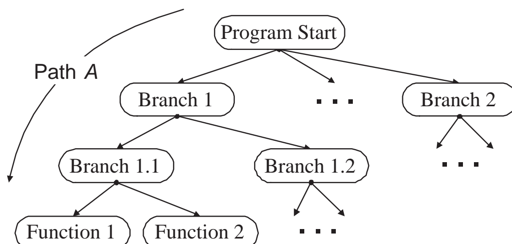


Figure 2: Tree structure of the program

Let the program be P and we consider any single path A. Let the path be broken down in $n$ basic blocks, $b_1, b_2, \cdots, b_n$ such that the control flow of the path A starts at $b_1$ followed by $b_2$ and then $b_3$, etc. The blocks are in encrypted form except the starting block $b_1$. The jumping code to the decryption routine is placed inside the basic blocks. The program controller, which implements the dynamic integrity verification, is stored at the end of the original program as illustrated in Figure 3.

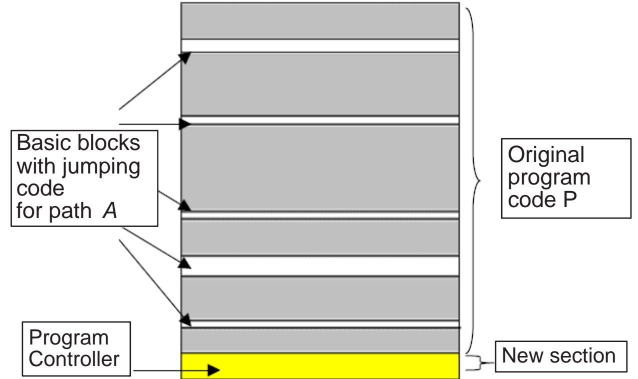


Figure 3: Structure of protected program

The entry point of the protected program is now set at the program controller. Once the initial state has been set up, the original program begins execution. Before the execution of $b_i$, calculate the hash value of $b_{i-1}, H_{i-1} = Hash(b_{i-1})$. We treat the hash value $H_{i-1}$ as the secret session key $K_{i-1}(K_{i-1} = H_{i-1})$ for the decryption of the block $b_i$, provided that the number of bits for $K_{i-1}$ is compatible with the encryption algorithm. If a hardware token was used, this can be implemented by using the $C_DeriveKey$ API function of PKCS #11 [12]. $C_DeriveKey$ can derive a secret key from a known data, $H_i$ in our case. In order to illustrate the algorithm, we take $n = 3$ in the following:

Algorithm: Multi-blocking integrity check (during program execution)

1. Before $b_2$ starts to execute, the program jumps to the program controller to calculate $H_1 = Hash(b_1)$, where $b_1$ is in plaintext.

2. The secret key $K_1$ is then derived from $H_1$, e.g. $K_1 = H_1$.

3. The second block $E(b_2)$ is decrypted by $K_1 : b_2 = D_{K1}(E(b_2))$.

4. The decrypted block is moved to its original place, followed by the execution of the decrypted codes.

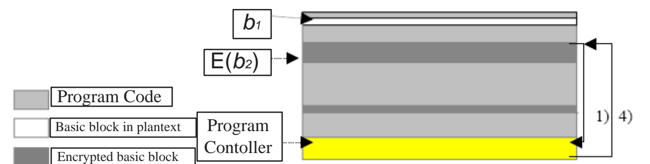


Figure 4: Program Progress 1 - 4 steps

5. Before $b_3$ starts to execute, the program jumps to the program controller to calculate $H_2 = Hash(b_2)$.

6. The secret key $K_2$ is derived from $H_2$, e.g. $K_2 = H_2$.
7. The third block $E(b_3)$ is decrypted by $K_2 : b_3 = D_{K2}(E(b_3))$.
8. The decrypted block is moved to its original place, followed by the execution of the decrypted codes.

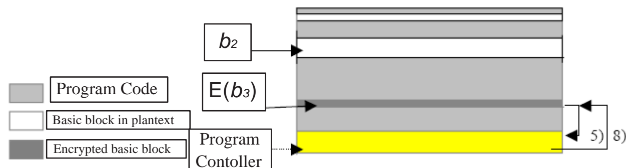


Figure 5: Program Progress 5 - 8 steps

In order to create different keys for encrypting different blocks, we use different hash values to achieve this property. There is no 'storage of keys' problem as the hash values are calculated dynamically during program execution. The above completes the treatment of a single path. For the whole tree structure program P, we can apply the algorithm on all paths inside the tree.

The adversary cannot modify the software statically as binary codes are in encrypted form after protection. For dynamic modification, suppose an adversary alters the running program in block $b_i$ which will produce a different hash value $H_i$. Before the execution of $b_{i+1}$, the hash value $H_i$ is not the proper decryption key $K_i$ for block $b_{i+1}$, the result from the decryption will then produce rubbish code. Due to the corruption of the next block, the program cannot continue to run properly and crashes. It is a great advantage that the program will not halt its execution immediately after code modification. When the tampered program crashes, the adversary will find it very difficult to trace back the exit point.

Using this multi-block hashing method, no hash value comparison is present and bypassing the checking is impossible. The scheme is constructed so that any program state is in a function of all previous states. Therefore, the program is guaranteed to fail if one bit of the protected program is tampered with. The point of failure also occurs far away from the point of detection, so that the adversary does not know how it has taken place.

### 3.4 Analysis of Multi-block Hashing Scheme and Enhancement

In contract to common hash value comparison schemes, this scheme does not use a single code block for integrity checking. This makes the adversary difficult to bypass the checking in the program.

To attack the scheme, the adversary may find out the hash value of each block by dynamic analysis. After finding out those hash values, he or she can replace the tampered hash value with the correct one during program execution and the program can decrypt the next basic block properly. However, the time taken by dynamic analysis is typically at least proportional to the number of instructions executed by the program at runtime. In other words, to attack our scheme, it takes a lot more effort than the attack for the previous schemes. In fact, we can further enhance the security of our approach in order to prevent the adversary to find out the hash values. One effective way to achieve this is to obfuscate the program codes. Some techniques are discussed in [6]. The use of multi-blocking encryption in our mechanism is, in fact, also one kind of obfuscation techniques.

On the other hand, we can also use the technique of code polymorphism [8] to prevent this problem, which means that the program code is mutated after each execution while preserving its semantics. Many computer viruses use this technique to prevent the anti-virus engines from finding them out. We use this idea to make our protection not noticeable by the adversary. One possible implementation is to mutate each basic block $b_i$ after it has been executed and thereby changing its hash value. We can use the new hash value to re-encrypt the next block $b_{i+1}$ and so on. This creates a new version of the same program with identical block decomposition. Even if the adversary can identify the hash values at the first time, those values cannot be used for the next execution of the program as the hash values have been mutated after the previous execution.

## 4 Experimental Results

In this section, we want to find out the overhead in terms of program size and execution time. We have implemented our multi-blocking encryption technique to a software application: gzip [13] is a compression utility designed to be a replacement for compress and the control flow is in a tree structure. The corresponding binary object code is analyzed to determine the number of basic blocks to be used and the blocks information is passed to the installation program, as shown in Figure 6.

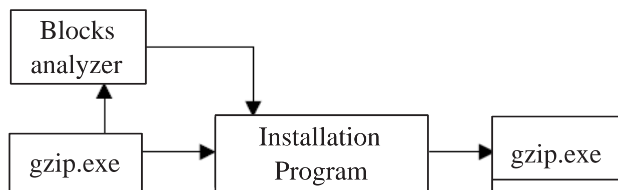


Figure 6: Installation of dynamic integrity verification scheme

We have applied the integrity verification scheme in one single path of the tree structure, which is a subset of the whole tree structure program. The program controller and information for each basic block are added to the end of the object code automatically after passing through the installation program. USB hardware token was used for encryption and hashing, which conforms to the PKCS #11 [12] standard. Those basic blocks are encrypted with AES and we use SHA-1 for hashing. The integrity verification scheme is then installed into the program and is ready to be run. The installation process is automatic such that it is error free. The experiments were conducted on a Pentium 1.5GHz CPU clock with 256MB RAM. We have used

two files to be compressed by the gzip program with the size 89.5KB and 1.01MB, respectively.

### 4.1 Program Performance

We want to find out if the multi-block hashing would impose unreasonable runtime overhead on the program. An experiment was conducted to measure the runtime overhead of multi-block hashing versus number of blocks used and the results are shown in Figure 7. We have identified 89 blocks for the experiments. The execution time of Windows programs may differ slightly every time the code is executed. We chose to find an average execution time for the gzip program. While there were many jumping between the original program and the program controller, the run time overhead was reasonable. The execution time of the original program was 0.0467s and 0.1367s for zipping the 89.5KB and 1.01MB file, respectively. It took only 1.07s for zipping the 1.01MB file with 70 blocks used. The overhead is reasonable even the number of blocks used is large.

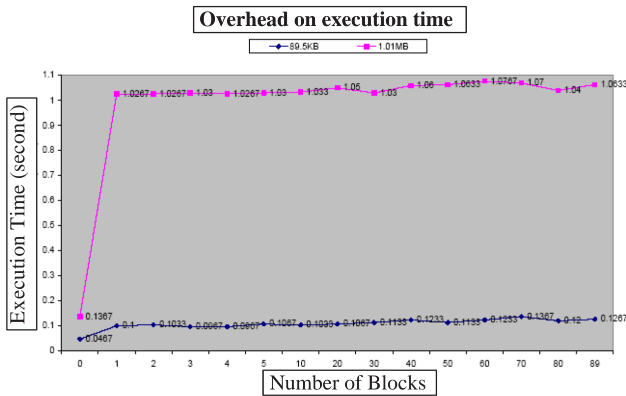


Figure 7: The average execution time of the protected gzip program with different number of blocks used

### 4.2 Program Size

The verification scheme installed gzip program size with different number of blocks installed was shown in Figure 8. The increase in size for storing information of each block is proportional to the number of blocks identified in the program.

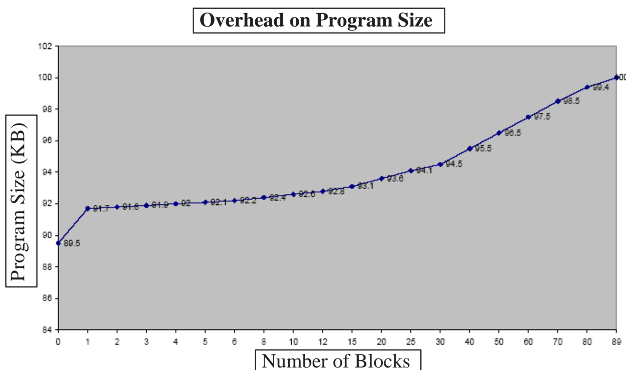


Figure 8: gzip program size with different number of blocks used

It can be shown that the size overhead for 30 blocks used is only 5.6% compared with the original program size, which is 89.5KB. It is relatively small as the program controller is written in low level assemble language which can be easily manipulated the object code of the gzip program.

## 5 Comparison with Previous Schemes

Recently, there have been active development on software dynamic integrity verification. Unlike Horne *et al.*'s dynamic self-checking mechanism [9], which consists of a number of testers that redundantly test for changes in the executable code as it is running and report modifications, our scheme detects the modification of codes by decrypting the next block code, not depending on any tester. Whether encrypted blocks can be decrypted properly, entirely depends on whether there is no modification on software. Otherwise, the software can not run properly, sometimes crashed, rather than exit in a traceable way. With this property, our scheme can prevent single check point failure attack.

Also, Chang and Atallah [3] proposed another approach based on a distributed scheme, in which protection and tamper-resistance of program code is achieved, not by a single security module, but by a network of (smaller) security units that work together in the program. These security units, or guards, can be programmed to do certain tasks (check summing the program code is one example) and a network of them can reinforce the protection of each other by creating mutual-protection. Although a group of guards are more resilient against attacks than a single branching instruction for comparing hash values, it only spreads out the single attack point into different locations of the program. In other words, although more complicated, bypassing instructions performed by the guards is still possible. Our scheme have similar inter-related structure, but achieved higher level security by preventing static code analysis attack.

In [4], Chen and Venkatesan proposed a novel software integrity verification primitive, Oblivious Hashing, which implicitly computes a fingerprint of a code fragment based on its actual execution. Its construction makes it possible to thwart attacks using automatic program analysis tools or other static methods. This new method verifies the intended behavior of a piece of code by running it and obtaining the resulting fingerprint. However, there is a practical constraint for binary-level code injection and if the adversary can locate the instructions for hash value comparison, bypassing is still possible. Since our multi-block hashing scheme decrypts necessary block during program execution, it can avoid run-time attack.

We compared the above three schemes on software dynamic integrity verification with our scheme, with respect to the ability against certain attacks, such as single check point failure attack, static code analysis attack, dump memory attack and run-time attack. The results illustrated in Figure 9.

| | Dynamic Self-Checking | Protection by Guards | Oblivious Hashing | Our Scheme |
|---|---|---|---|---|
| Single Check Point Failure Attack | -- | -- | O | O |
| Static Code Analysis Attack | O | -- | O | O |
| Dump Memory Attack | O | O | O | O |
| Run-time Attack | X | X | X | -- |

O: Completely against certain attack
--: Partially against certain attack
X: Vulnerable to certain attack

Figure 9: Comparison with Previous Schemes

## 6 Conclusion and Future Work

In this paper, we have presented a dynamic software integrity verification scheme that made use of multi-block encryption technique. The objective is to prevent an adversary from modifying the software. The scheme is different from previous integrity verification techniques in that no hash values comparison is required. This makes the adversary difficult to detach the checking from the program. If the program was altered, the program will not terminate in a traceable way with the help of debugging tools or other static analysis methods.

We also propose the use of code polymorphism to enhance the security of our scheme in case the adversary can trace out all hash values of each block. The coding of the basic block is mutated every time after it is executed and therefore its hash value is also changed. Even if the adversary can find out the hash values, those values cannot be used for the next execution of the program since the hash values have mutated during the previous execution.

One disadvantage of our scheme is that it only applies to programs with a tree-like control flow. The scheme cannot handle the case when there are several entry points entered to a single block. How to enhance our scheme to make it work in programs with a network-like control flow is an interesting topic for future study.

The dynamic integrity verification scheme can be applied to all Win32 PE format files like EXE and DLL. The experimental results showed that the size overhead is relatively small and the program execution overhead is reasonable.

## References


[1] M. Antonio, P. Ernesto, *An Efficient Software Protection Scheme*, Proceedings of the 16th international conference on Information security: Trusted information: the new decade challenge, pp. 385-401, 2001.

[2] D. Aucsmith, *Tamper Resistant Software: An Implementation*, Information Hiding, First Int'l Workshop, R.J. Anderson, (Ed.), pp. 317-333, May, 1996.

[3] H. Chang and M. J. Atallah, *Protecting Software Code by Guards*, Proc. 1st ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, Pennsylvania, Springer LNCS 2320, pp. 160-175, November, 2001.

[4] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, *Oblivious Hashing: A Stealthy Software Integrity Verification Primitive*, 5th International Workshop on Information Hiding, Noordwijkerhout, The Netherlands, Springer LNCS 2578, pp. 400-414, 7-9 October, 2002.

[5] C. Collberg and C. Thomborson, *Software watermarking: Models and dynamic embeddings*, In Principles of Programming Languages, San Antonio, TX, pp. 311-324, January, 1999.

[6] C. Collberg, C. Thomborson, *Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection*, IEEE Transactions on Software Engineering, Vol. 28 No. 6, pp. 735-746, 2002.

[7] C. Collberg, C. Thomborson, and D. Low, *A taxonomy of obfuscating transformations*, Technical Report 148, University of Auckland, 1997.

[8] M. J. Granger, C. E. Smith and M. I. Hoffman, *Use of Pseudocode to Protect Software from Unauthorized Use*, United States Patent 6,334,189 B1 12/25/2001.

[9] B. Horne, L. Matheson, C. Sheehan, 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.

[10] B. Schwarz, S. Debray, G. Andrews, *Disassembly of Executable Code Revisited*, Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), pp. 45-54, October 29-November 01, 2002.

[11] Hardware token example: ikey – http://www.rainbow.com/products/ikey/index.asp

[12] PKCS #11 - Cryptographic Token Interface Standard http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/

[13] The gzip compression program: http://www.gzip.org/