Towards Better Software Tamper Resistance

Hongxia Jin, Ginger Myles, and Jeffery Lotspiech

IBM Almaden Research Center
San Jose, CA, 95120
{jin,gmyles,lotspiech}@us.ibm.com

Abstract. Software protection is an area of active research in which a variety of techniques have been developed to address the issue. Examples of such techniques include code obfuscation, software watermarking, and tamper detection. In this paper we propose a tamper resistance technique which provides both on and offline tamper detection. In our offline approach, the software dynamically detects tampering and causes the program to fail, protecting itself from malicious attacks. Additionally, during program execution an event log is maintained which is transmitted to a clearing house when the program is back online.

Keywords: Software protection, tamper detection

1 Introduction

The protection of software from hackers is a major concern for many industries. Foremost are the software developers themselves who are concerned about the loss of revenue due to piracy. Additionally, the music and movie industries are worried about the software which protects their copyrighted material. Once the protection software has been circumvented the content can be freely copied. Content protection technologies can only work effectively when the software that implements them is protected. In other words, their implementations are tamper resistant. The development of tamper resistant technologies, especially software tamper resistance has become a growth industry.

To illustrate the usage model consider IBM's Electronic Media Management System (EMMS) [5] for selling music online. Under this business model, a user buys a software media player which contains an embedded Digital Rights Management system. Music is bulk-encrypted and can be downloaded from the Web to the user's hard drive. The consumer's software connects with the clearing house and gets the decryption key for the music purchased. The music will only play using the correct decryption key. Similarly, it is conceivable to envision a movie studio giving away promotional DVDs which include specific usage criteria. Two possible usage scenarios include full movie viewing only after a fee or allowing complete viewing after a specified time period. The ability to enforce access rights to the copyrighted content is the key to the success of these types of business models.

J. Zhou et al. (Eds.): ISC 2005, LNCS 3650, pp. 417–430, 2005.

Tampering with the software is usually done through reverse engineering. Software tamper resistance, which refers to the art and science of protecting software from unauthorized modification, distribution and misuse, provides a powerful way to protect software from such activities. In this paper we propose a software protection technique directed at client-side software running on a potentially hostile host. Our approach provides both on and offline tamper detection. In the offline environment the software dynamically, self-detects tampering and causes the program to fail. As the program executes an event log is maintained. During online execution the log is transmitted to a clearing house where it is analyzed for evidence of tampering.

2 Background

Many techniques have been developed to solve the problem of protecting the host against the potentially hostile actions of the software it is running. Relevant work in this area includes Java Security [2] and Proof-carrying Code [8]. To combat such an attack requires restricting the actions of the malicious program. Tamper resistance addresses the opposite concern, running trusted code on untrusted hosts. It should be noted that it is much more difficult to combat a malicious host than it is to combat a malicious program. Since the host has full control over the software's execution, it is generally believed that given "enough" time, effort, and/or resources a sufficiently determined attacker can completely break any piece of software.

The issue of software protection can be addressed from either a software or hardware-based approach. Hardware-based techniques generally offer a higher level of protection but at the cost of additional expenses for the developer and user inconvenience. Additionally, software is purchased and distributed over the Internet which makes the use of certain hardware-based techniques, such as dongles or smartcards, infeasible. Tamperproof CPUs are another hardware-based solution, however this type of hardware is not widely used.

Software-based approaches address the issues of cost and user convenience but the protection is usually easier for an adversary to circumvent. One technique to prevent tampering is to increase the difficulty for hackers to attack the software. Several techniques have been proposed in this direction. Code obfuscation [1, 6] attempts to transform a program into an equivalent one that is more difficult to understand through static and dynamic analysis. The major drawbacks of all obfuscation approaches are that by necessity they are ad hoc and often introduce additional overhead.

Another software-based technique, which can provide provable protection against tampering, is to encrypt programs and execute them without the need for decryption. Sander and Tschudin proposed one such technique [9]. Their technique relies on identifying specific classes of functions, namely polynomials and rational functions. Since not all programs contain such functions the technique has limited applicability.

Customization can protect a program from tampering by making different copies of the software for different users. Distributing alternate versions can

better defend against "break-once, break everywhere" attacks. When one version of a program is broken and its patch is published, other users cannot exploit the patch to break their copy.

Software-based techniques also include tamper detection and tamperproofing. In order to detect tampering, it may be necessary for the software to leave behind evidence during execution. The evidence is examined later to provide evidence of tampering and to determine the appropriate course of action. In this paper we present a scheme that seamlessly combines several of the above described approaches to provides both on and offline tamper detection.

3 Design Objectives

Most of the research in the area of software protection conducted thus far is ad hoc without provable security guarantees. The area of tamper resistance is no different, and is seen more as a black art than a science. Standards to measure the effectiveness of tamper resistance techniques do not currently exist. In order to push towards a standardized criteria for tamper resistant algorithms, we outline our design considerations below.

The goal of any tamper resistance technique is to prevent an adversary from altering or reverse engineering the program. Overall, a good technique should be comprehensive, stealthy, flexible and have low overhead. The ideal objective is to prevent modifications in the program. However, a more realistic objective is to make modification difficult, detect it and take action against it. Below are objectives to defend against various attacks.

- The technique should be able to detect small changes, even a single bit, in essential portions of the program.
- The use of a debugger or similar tools should be detected regardless of whether or not the debugger relies on modifying the code.
- Tampering should be detected in a timely manner so that temporary modifications are not missed. A dynamic attack can make temporary modifications to the program but restore it back to normal after completion.
- The response mechanism should be separate from the detection mechanism. This will increase the stealth of the entire mechanism and permit flexible responses based on the type of tampering detected.
- The detection mechanism should be stealthy and obfuscated to limit static attacks.
- A variety of detection mechanisms should be used throughout the program to increase the level of analysis required to detect the protection.
- It is preferable that the detection mechanism is customized for different copies of the program. This aids in defending against automated systematic attacks.
- The detection mechanism should provide complete and comprehensive coverage.
- The response mechanism should be stealthy and/or obfuscated. Ideally it should blend in with normal program behavior to make it hard to detect.

- The response mechanism should be customized to different copies of the program. Understanding and disabling one would not disable others.
- If using installation patches, the patch should be stealthy and not reveal information about the mechanism.
- If the program is customized by user, the scheme must consider defending against collusive attacks.
- To make it difficult for hackers to understand/disable the scheme, a single point of failure should be avoided.

4 Design Assumptions

The proposed tamper detection techniques make the assumption that an attacker will make at least one initial failure before the software is completely understood. Such an assumption has limitations when dealing with professional hackers who are equipped with extensive computing resources. Given the proper resources an attacker can completely or partially replicate the state of the program execution to another machine. Of course, finding useful information from the large number of states recorded is no easy job. In fact, it may even be an intractable task. However, because it is known that attacks are often performed in a simulated and instrumented environment, the proposed techniques incorporate features which limit the effectiveness of the attack tools. This has the effect of limiting the weaknesses associated with our assumption in many attack scenarios.

5 Proposed Tamper Detection Technique

The proposed tamper detection technique consists of two united parts to provide software protection in both on and offline environments. The two techniques are based on the central underlying theme of key evolution and integrity checks. Since the available resources vary in the on and offline environments the two approaches uniquely build from the common base. The online technique records execution events in a tamper resistant log thereby producing an audit trail for anomaly detection. The offline version is able to use the execution events to self-detect abnormalities.

A key aspect of the scheme is the use of integrity checks. An integrity check is an inserted section of code used to verify the integrity of the program and to detect active debugging. Integrity checks are triggered during software execution. For example, one of the integrity checks could choose a block of code and calculate its checksum. If the hacker attempts to store breakpoints or to modify the code, even if the modification is very slight, the checksum will be wrong. When trying to detect the presence of a debugger, the elapsed time of executing from one point to another can be used as an integrity check. These simple integrity checks are just for illustration purpose. In practice a variety of stealthy integrity checks are used. Often these checks are customized to address the specific requirements of the application. Due to the nature of integrity checks they are often regarded as trade secrets. Publishing details of the exact checks used

would decrease the potency. This is true of most techniques aimed at providing tamper resistance.

5.1 Online Tamper Detection

The online tamper detection portion of the scheme is based on a technique we previously developed [4]. In this section we provide a summary of the technique so that it is clear how the on- and offline schemes are united to form a stronger tamper detection mechanism. To protect the application using the online scheme, integrity check code is embedded throughout the original application. As the program executes the results of the integrity checks are recorded in an event log. At periodic intervals the log is transmitted back to a clearing house where the entries are examined for evidence of tampering.

The event log plays an important role in the detection scheme. Ideally, the integrity check logging process would be accomplished in a stealthy manner which is undetectable by the attacker. Unfortunately such an event is unlikely in a scenario where the attacker has full access to the software. Therefore, precautions must be taken to ensure that an attacker cannot damage the entries.

To this end we have developed a tamper resistant method for logging the integrity check results. The basic idea is that the log entries are dependent on a key that evolves through a one-way function. Because the evolution is one-way the attacker is unable to use the current information to forge previously recorded log entries. Figure 1 illustrates one possible approach for the tamper resistant log [4].

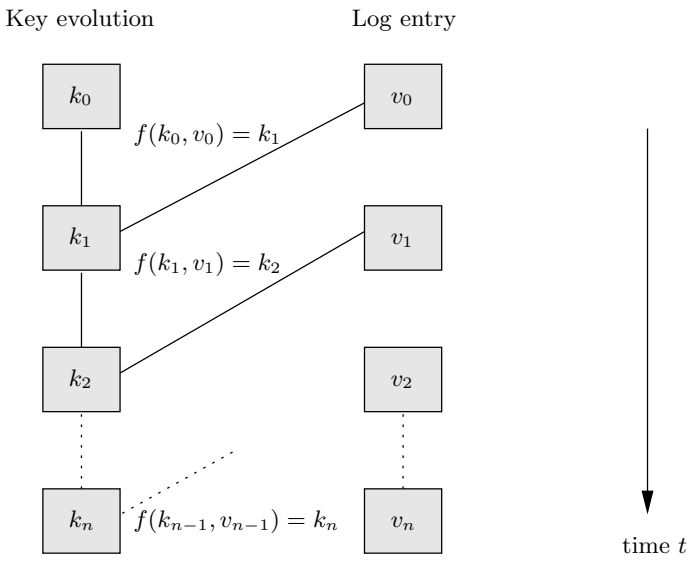

Fig. 1. A way to perform tamper resistant logging

To implement the tamper resistant log the one-way function f uses both the current key k_i and the current integrity check value v_i to generate a new key k_{i+1}.

$$k_{i+1} = f(k_i, v_i)$$

Every time a new key is generated, the previous key is destroyed. This limits the information available to the attacker at any one instant. As the program executes the series of integrity check values are recorded. The log together with the last calculated key k_n are transmitted back to clearing house. If the software is modified some integrity check value v_i will differ from what is expected. The resulting effect is that the key evolution will be incorrect. When k_n is transmitted, the key evolution calculated by the clearing house will differ from the submitted value. If the incorrect integrity check value v_i is not modified in the log, it is clear evidence of the tampering.

The integrity checks can be embedded anywhere in the original application, however, if the points are chosen such that they are encountered along all execution paths only the final key k_n needs to be transmitted. Using such a placement the clearing house knows the correct value for each integrity check. With this knowledge the clearing house can evolve the key using the initial key. If the submitted key differs from the calculated key tampering has been detected. This option enables a minimal log size.

After verification, if no tampering is detected, the program can proceed as usual and the key will continue to evolve. However, if tampering is detected the clearing house can take appropriate measures, such as warning the user about such activity, blocking future content, or taking legal action.

The online tamper detection scheme has a few limitations. First to detect tampering it is required that the attacker contact the clearing house. This will not occur if the attacker is aware of the tamper detection mechanism. This leaves the attacker with a functioning piece of software and we have not detected the tampering. Additionally, there is the chance the log is forged making it impossible for the clearing house to detect the tampering. The offline scheme addresses these issues to improve the tamper detection capabilities.

5.2 Offline Tamper Detection

The same key evolving mechanism can be used as a basis for offline tamper detection. The key evolution can be used in controlling program execution and ultimately cause the program to fail. There are a multitude of ways key evolution can be utilized to achieve tamper detection/tamperproofing in software. For example, a key value can be transformed into a valid constant variable that will be used later in the program. If tampering occurs, the key generated will be invalid and the transformation will yield an incorrect value for the constant variable. This will ultimately lead to program failure. Of course, more complex and obfuscated techniques can be designed around key regulated program execution. For example, a more expensive tamperproofing approach is to encrypt portions of the code using a valid key at a particular place in the program. If

tampering occurs an incorrect decryption key is used. We have devised a tamper detection technique which is less costly than the use of encryption but still offers the desired tamper detection benefits. We call this technique *branch-based tamper detection*. The branch-based tamper detection is similar to a software watermarking technique we proposed [3]. Both schemes use key evolution and a branch function to control execution. However, the watermarking scheme uses the key as the program's fingerprint and the tamper detection scheme uses the key to detect program alterations.

5.3 Branch Based Tamper Detection

The basic idea of the branch-based tamper detection algorithm is centered around the use of a branch function similar to the one proposed by Linn and Debray to disrupt static disassembly of native executables [7]. The original obfuscation technique converted unconditional branch instructions to a call to a branch function inserted in the program. The sole purpose of the branch function is to transfer the control of execution to the instruction which was the target of the unconditional branch. The branch function can be designed to handle any number of unconditional branches. Figure 2 illustrates the general idea of the branch function. To increase the versatility of the branch function we have devised an extension which makes it possible to convert conditional branches as well. When this idea is applied to the x86 instruction set all `jmp`, `call`, and `jcc` instructions can be converted to calls to a single branch function. In order to provide tamper detection for the entire application the branch function is enhanced to incorporate an integrity check and key evolution. Multiple integrity check branch functions are incorporated to develop a self-monitoring check system for the entire program.

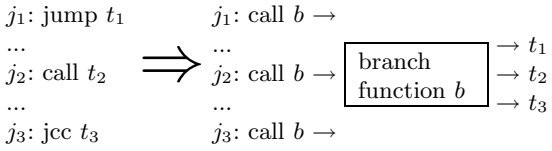

Fig. 2. Branch instructions are converted to a call to a branch function which returns to the instruction which was the target of the branch.

Enhanced Branch Function The original branch function was designed simply to transfer execution control to the branch target. In addition to the transfer of control, the *integrity check branch function* (ICBF) incorporates an integrity check and key generation into the target computation. The ICBF performs the following tasks:

- An integrity check producing the value v_i.
- Computation of the new key k_{i+1} using v_i and the current key k_i, $k_{i+1} = g(k_i, v_i)$.
- Identification of the displacement to the target via $d_{i+1} = T[h(k_{i+1})]$, where T is a table stored in the data section and h is a hash function.
- Computation of the return location by adding the displacement d_i to the return address.

Through the enhancements the ICBFs can provide tamper detection for the entire program.

Tamper Detection Transformation The tamper detection mechanism is incorporated into the program by injecting multiple ICBFs into the program and converting a selection of branch instructions to calls to the ICBFs. The transformation occurs in four phases. In the first phase the set of to be converted branches is selected, $\{b_1, ..., b_n\}$. Special care must be taken in selecting which branch instructions are converted. The branch instructions used in any given function must reside on a path that will be traversed every time the function executes. Without imposing this constraint an irregular key evolution will occur resulting in an incorrect return location and improper program behavior. In addition, because a new key is generated every time the branch function is executed the branch instructions cannot be part of a non-deterministic loop. The usable set of branches can be identified through data-flow analysis.

In the second phase a mapping is constructed between the set of branches and the ICBFs.

$$\theta : \{b_1, ..., b_n\} \rightarrow \{ICBF_1, ..., ICBF_k\}$$

This mapping is then used in phase three when the branches are replaced by calls to the appropriate ICBF. In the final phase the displacement table is constructed. For each branch replaced a mapping is maintained between the calculated value k_i and the branch, target displacement d_i.

$$\phi = \{k_1 \rightarrow d_1, ..., k_n \rightarrow d_n\}$$

ϕ is used in this phase to construct the displacement table T. The first step is to construct a hash function such that each value k_i maps to a unique slot in the table. By using a minimal perfect hash function the table size can be minimized.

$$h : \{k_1, ..., k_n\} \rightarrow \{1, ..., m\}, n \leq m$$

Based on h the table is created and added to the data section of the binary.

$$T[h(k_i)] = d_i$$

Tamper Detection Mechanism Highlights Through the use of multiple integrity check branch functions a check system can be established which enables

self monitoring of the entire program. The check system could be configured such that one integrity check verifies that another has not been modified or removed.

In our scheme the software dynamically detects tampering through the computation of k_i. If either the key or the integrity check are altered, an incorrect slot in the table will be accessed. Since the slot is wrong, an incorrect displacement will be added to the return address. Upon function return an incorrect instruction will execute eventually leading to program failure which is the desired result for tamper detecting software.

The robustness of many tamper detection techniques suffer because the detection mechanism relies on a comparison between the calculated value and the expected value. This is considered a weaker form of detection since it is often easy for an attacker to remove the check. In the branch-based tamper detection scheme the calculated value is directly used in controlling the execution of the program. Thus eliminating an important vulnerability.

Strength Enhancing Feature It is possible to further enhance the strength of the tamper detection algorithm through the use of indirection. Added levels of indirection increase the amount of analysis required by an attacker for program understanding. Further indirection can be incorporated by rerouting all calls to the ICBFs through a single super branch function which transfers execution to the proper branch function.

5.4 Key Protection

Both of the proposed techniques suffer from the same vulnerability. In each algorithm an initial key is required to begin the key evolution process. In the branch-based technique the same initial key is used each time the program executes. When the online version is used alone the original initial key is not required each time the program executes. Instead the key which was generated last can be used. Without protection for the initial key the additional strength provided through the one-way function is lost.

One such technique is to use an array of cells. Each cell in this array contains the key k_0 encrypted with a valid key that the program could generate during execution. More specifically, the key k_0 is concatenated with a verification string, e.g., "DEADBEEF", and then encrypted with each valid key, including k_0 itself. When the program starts, it first decrypts each encrypted cell. If the last evolved key is valid, then one of the decryptions will show the verification string in the decrypted buffer. The decrypted buffer will also reveal the value of k_0.

During execution, the key evolves and the new key overrides the old key. If the program crashes because of innocent customer error, the last key it calculates should be valid. Using that valid key the initial key can be obtained from the encrypted cell and the program can be restarted correctly. On the other hand, if the program crashes because of tampering, it will generate an invalid key. Using this invalid key, the decryption of the encrypted cells cannot end up with the correct initial key thus the program cannot restart. In the online protection

mechanism, to solve the problem the user can contact the clearing house. The actions that the clearing house take can vary depending on the business scenario. It can mark the user and pay more attention to this particular user in the future. When the occurrence of the same incidence exceeds some threshold, it becomes more confident that the user is tampering with the software and the user can be disconnected from the service network. Under the offline technique the user is left with non-functioning software.

5.5 Uniting the On- and Offline Techniques

The strength of a protection scheme can be improved when multiple protection techniques can be tightly coupled. We can improve the tamper detection capabilities by making use of the strengths from both the on and offline versions. The united version will use the branch-based tamper detection as well as the tamper resistant log. Additionally, because periodic connections will be made to the clearing house, the initial key used by the branch-based mechanism can be reset to a new value. This will also require that a patch be applied to update the values in the displacement table. Such a modification will require an attacker to restart any analysis conducted thus far. Of course, because we can choose to weave integrity checks which overlap, it is possible that different integrity checks are triggered for different executions. For example, the updated new key can be used to decide what integrity checks will be triggered. Again, such an update will require an attacker to restart a new analysis.

6 Analysis of the Scheme

The goal of any tamper detection technique is to prevent an adversary from altering or reverse engineering the program. Based on this criteria we have evaluated the robustness of the technique based on its ability to withstand a variety of automated and manual attacks.

One of the most common forms of automated attack is code obfuscation. Through the use of the system of integrity check branch functions a program is able to self-detect semantics-preserving transformations. We applied a variety of transformations to verify that the tamper detection mechanism behaved as expected. In each case the protected application failed to function correctly after the obfuscation had been applied.

A common manual attack is to inspect the code in order to locate and remove a license check. When a program has been protected using branch-based tamper detection, successful removal of the license check requires the attacker to remove the entire tamper detection system. Such an attack requires unravelling the table and replacing all of the calls with the correct branch instruction and displacement, otherwise the alteration will be detected. To unravel the table and determine the correct instruction requires extensive dynamic analysis which in many cases may be prevented by the integrity checks. For example, the use of a debugger could be self-detected and lead to incorrect program behavior. Baring

the use of a completely secure computing device, guaranteed protection against manual attacks is impossible. All that we can hope is that the analysis required is extensive enough that an attacker finds it too costly.

The robustness against reverse engineering is partially based on the number of converted branches. Since the algorithm requires the branches to be on a deterministic path the number of usable branches is being limited. Through analysis of a variety of different applications, we found a satisfactory number of conditional and unconditional branch instructions. To illustrate Table 1 shows the total number of branches and the number of usable branches in the SPECint-2000 benchmark applications. By additionally using conditional branches we are able to significantly increase the number of usable branches. While the removal of the tamper detection capabilities is not impossible, the manual analysis required to accomplish the task is extensive.

Program	Total Branches	Usable including conditionals	Usable excluding conditionals
gzip	2843	464	170
vpr	5814	1153	674
gcc	28136	4886	3056
mcf	2028	290	89
crafty	3340	496	178
parser	5628	864	522
gap	18999	1942	1027
vortex	16144	3462	1049
bzip2	2354	457	211
twolf	4397	729	429

Table 1. Total number of branches versus the number of usable branches in the SPECint-2000 benchmark suite applications.

The tamper detection technique also inhibits the adversary's ability to reverse engineer the program. By replacing conditional and unconditional jumps the obvious control flow of the program has been removed. The tamper detection is based on information only available at runtime. This eliminates the use of static analysis tools. In order to completely reverse engineer the program the attacker will have to dynamically analyze the program which will be significantly inhibited by the integrity checks.

In our scheme, the software can be distributed in a traditional manner. If customization at the user level is required the software will be non-functional until the user registers it with the company. At that time a patch file is distributed which will create a fully functional program. The patch will contain the initial key in the form of an array of encrypted cells and the displacement table.

The most crucial attack on a customized application is the collusive attack. This occurs when an adversary obtains multiple differently customized programs and is able to compare them. The branch-based tamper detection scheme is highly resistant to the collusive attack. The only difference between two cus-

tomized programs is the order of the values in the table. Thus, an attacker would have to examine the data section in order to even notice a difference.

The algorithm is still susceptible to dynamic collusive attacks but some of those attacks can be warded off through the use of integrity checks which recognize the use of a debugger and cause the program to fail. In a dynamic attack the only difference the adversary is going to notice is the value of the key that is generated at each stage which will ultimately yield a different table slot. In order for an adversary to launch a successful collusive attack extensive manual analysis will be required to remove the detection mechanism.

The detection and response mechanisms are stealthy. Once the tampering is detected the program will behave improperly and ultimately fail. Even though the detection is immediate, the response is separated and delayed. This increases the stealthiness and makes it difficult for the attacker to identify the point of failure.

7 Experimental Results

It is not hard to imagine that when using our scheme the size of the program will increase and that there will be a degradation in performance. Even though we suggest that it is desirable to apply a variety of tamper detection mechanisms, we have only performed an experimental evaluation on the branch-based technique

We have created a prototype implementation for Windows executable files. The tamper detection capabilities are incorporated by disassembling a statically linked binary, modifying the instructions, and then rewriting the instructions to a new executable file. To evaluate the overhead we used the SPECint-2000 benchmark suite applications. We were unable to use eon and perlbmk because they would not build. Our experiments were run on a 1.8 GHz Pentium 4 System with 512 MB of main memory running Windows XP Professional. The programs were compiled using Microsoft's VisualStudio C++ 6.0 with optimizations disabled. The execution times reported were obtained through five runs. The highest and lowest values were discarded and the average was computed for the remaining three runs.

As can be seen in Table 2 very little performance overhead is incurred by the additional calls and integrity checks. The unprotected benchmark application *gcc* did not execute properly on the reference inputs so we were unable to obtain performance information suitable for comparison with the other result. However, when run using the test data no significant slowdown was observed.

The majority of the space cost incurred by the branch-based scheme is based on the size of the integrity check branch functions and the displacement table. Additionally, any difference between the converted branch and the call instruction sizes will contribute to the size of the protected application. Table 3 shows the effect incorporation of branch-based tamper detection had on the size of the benchmark applications. For most of the applications the size increase was minimal. *gcc* was most significantly impacted but it was also the application in which the greatest number of branches were converted. A technique to minimize

Program	Execution Time (sec)		
	Original (T_0)	Protected (T_1)	Slowdown (T_1/T_0)
gzip	435.52	435.52	1.00
vpr	479.12	480.62	1.00
mcf	563.07	562.55	1.00
crafty	326.96	326.40	1.00
parser	519.31	588.34	1.13
gap	292.20	292.01	1.00
vortex	316.22	316.66	1.00
bzip2	743.18	739.82	0.99
twolf	912.43	922.84	1.01

Table 2. Effect of tamper detection mechanism on execution time.

the size impact is to use a perfect hash function in assigning the slots in the displacement table. Our implementation did not use a perfect hash function thus the results could be improved.

Program	Program Size (KB)		
	Original (S_0)	Protected (S_1)	Increase (S_1/S_0)
gzip	100	104	1.04
vpr	212	252	1.19
gcc	1608	2604	1.62
mcf	64	68	1.06
crafty	316	320	1.01
parser	184	188	1.02
gap	660	780	1.18
vortex	608	660	1.09
bzip2	88	96	1.09
twolf	316	332	1.05

Table 3. Effect of tamper detection mechanism on program size.

8 Conclusion

In this paper we describe a novel approach to software tamper detection which incorporates both an on and offline techniques to increase robustness. It includes copy-specific customization, obfuscation, and dynamic self-checking. Our technique is an improvement over previous techniques in that the software is able to dynamically self-detect alterations and cause program failure, protecting itself from malicious attacks. The self-validating mechanism embedded in the program can substantially raise the level of tamper resistance against an adversary with static analysis tools even if they have knowledge of our algorithm and some implementation details.

The prototype demonstrates that the technique is robust against various types of automated and manual attacks which makes it a viable protection mechanism for software running on a potentially hostile host. The space cost associated with the technique is a very small percentage of the size of the program, especially for large programs. Additionally, the mechanism had no adverse effects on the performance of the benchmark applications.

As part of our future work, we would like to eliminate the requirement in the branch-based technique that the same initial key be used each time the program is executed. Additionally, we would like to relax the branch selection requirement. We will continue to assume that hackers tumble first before they succeed and our scheme will hopefully detect the tampering by then. However, if the key generation points can be chosen more randomly rather than having to be on deterministic path, then even if attackers capture the branch trace once, they cannot use it again for other input data. We believe this can further improve the strength of the scheme.

References

[1] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, 1998.
[2] S. Fritzinger and M. Mueller. Java security, 1996.
[3] G.Myles and H. Jin. Self-validating branch based software watermarking. In *Information Hiding Workshop*, June, 2005.
[4] H.Jin and J.Lotspiech. Proactive software tamper detection. In *Information Security Conference*, volume LNCS 2851, pages 352–365, 2003.
[5] IBM. Electronic media management system.
[6] D. Libes. *Obfuscated C and Other Mysteries*. Wiley, 1993.
[7] C. Linn and S. Debray. Obfuscation of executable code to improve resistence to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 290–299, 2003.
[8] G. Necula. Proof carrying code. In *Twenty Fourth Annual Symposium on Principles of Programming Languages*, 1997.
[9] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, 1998. Springer-Verlag, Lecture Notes in Computer Science 1419.