A Survey of Binary Tamper-resistance Techniques for Software Protection

Xianya Mi[1, a], Yi Zhang[1, b], Baosheng Wang[1, c]

[1] College of Computer, National University of Defense and Technology, Changsha, 410073, China

[a]email: mixianya@126.com, [b]email: zhangyi@nudt.edu.cn, [c] bswang@nudt.edu.cn

Keywords: Software protection; Tamper-resistance; Check-respond mechanism

Abstract. Software safety issues have become severe problems in modern information industry. Binary tamper-resistance techniques as an important means for software protection, have become a worldwide popular research topic. In this paper, we firstly introduce basic concepts of software cracking and protecting, aiming at the main targets of tamper-resistance, which is precisely detecting abnormal modifying behaviors beyond the range of normal functions. Secondly, we analyze various kinds of tamper-resistance techniques based on check-respond mechanism, which can react to modifications properly and effectively. Finally, we categorize different implementations and summarize their superiorities and limitations with an evaluation system. We also propose our expectations of the development prospect on tamper-resistance techniques.

1. Introduction

As software cracking techniques developing rapidly, the safety issues of software industry have become severe problems. It’s increasingly difficult to protect software intellectual property. With software copies, attackers can easily understand and modify binary code using different tools, so that they can crack software and use it without legal permission. Binary tamper-resistance is one of the most important software protection methods and has become a research hotspot. In this paper, we study the basic principles and implementation schemes of tamper-resistance. We also analyze and evaluate different mainstream protection mechanisms based on tamper-resistance and propose our prospective on the application and expectations of this technique.

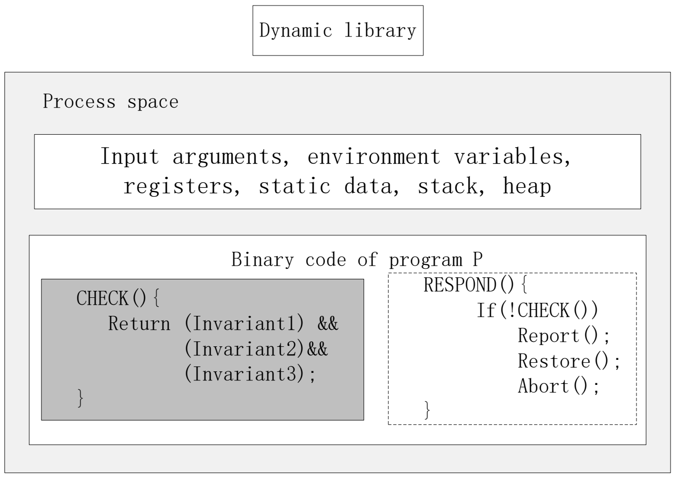
1.1. Software cracking and protection

After purchasing binary code, attackers can crack the software through the following five steps: analysis, tampering, piracy, automation, publication [1]. The basis of cracking is software analyzing, which can be done with reverse engineering using static or dynamic techniques. The key point of illegal usage is tampering, which means attackers will modify essential parts to achieve their goals of uncompensated use and limitations break-through. There are two kinds of illegal usage: unauthorized usage and illegal piracy. To reduce the work of cracking process, attackers often develop professional tools for automatically cracking, which is called automation. After all the work above, attackers will release the cracked software and make profit.

1.2. Basic concepts of tamper-resistance

Attackers use software tampering after analyzing binary code. There are three methods to realize effective tampering: 1) delete some old instructions and add new ones before the program to be executed; 2) delete some old instructions and add new ones while the program is running; 3) using emulations and debuggers to impact running behavior of the program. The first method can be defined as static tampering, which directly modifies binary code. The latter two can be categorized as dynamic tampering, which has similar features as debugging. To sum up, the ultimate goal of software tampering is to delete usage checking process, eliminate user marking fingerprints and add software functions.

The goal of tamper-resistance techniques is to detect modification behavior on the program and properly react to it. There are two essential principles: 1) Precise detection of abnormal modifying behavior; 2) proper and effective reaction of this behavior. Common tamper-resistance techniques are mostly implemented based on the two principles mentioned above and developed a mature method called check-respond mechanism. Using this mechanism, a complete tamper-resistance system should consist of two parts: CHECK and RESPOND. As shown in Figure 1, CHECK function tests if there are abnormal modifying behaviors and returns the results, while RESPOND function use these results to decide whether react to this behavior and stop it.


Dynamic library
Process space
Input arguments, environment variables, registers, static data, stack, heap
Binary code of program P

```
CHECK(){
    Return (Invariant1) &&
           (Invariant2)&&
           (Invariant3);
}
```

```
RESPOND(){
    If(!CHECK())
        Report();
        Restore();
        Abort();
}
```

Fig.1. A complete tamper-resistance system infrastructure.

2. Software tamper-resistance implementation

According to different implementation ideas of CHECK and RESPOND functions, software tamper-resistance can be divided into four categories: 1) tamper-resistance method based on protection guards network; 2) tamper-resistance method based on oblivious hashing; 3) remote tamper-resistance method; 4) tamper-resistance method based on monitoring running environment.

2.1. Protection guards network

Hoi Chang et al. [2] establish a protection network composed of code segments called guards. There are two kinds of guards in the network. One is checksum function to compute the hash value of some code and verify its integrity, to check if the code has been modified. If the checksum guard finds out the modification, it will respond to this situation and trigger some behaviors like disenabling the software functions or interrupting the program process. This kind of guard is used to check integrity. The other is repair function to restore the damaged code and data to the original version. To achieve this functionality, the original code should be backed up somewhere else. When modification happens, the original code is copied from there. This kind of guard is used to maintain self-treatment. If an attacker wants to destroy a protection guards network, he must destroy all the guards at one time, otherwise the tampering will fail because the guards rely on each other tightly, which makes it difficult for attackers to crack the software. There is another advantage of this scheme, which is the combination of several guards can allow complex implementation of different levels of protection intensity. For example, as shown in Figure 2, there is a protection network of five guards, i.e. G1, G2, G3, G4, G5. C1 and C2 are protected code segments. We can see the complex relations of these guards and code segments.

There are also some limitations of this implementation. Checksum functions only examine the other code in the program, which is not enough because the checksum functions should also be examined, otherwise the attackers can locate these guards and tamper them. However, if we try to make every function examined, the network will become very complex due to dependencies and is not easy to implemented.

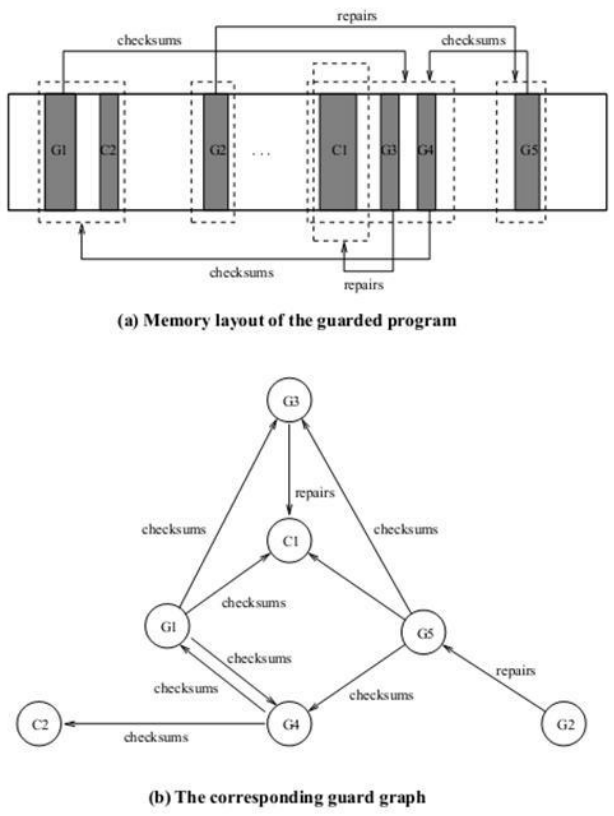


Fig.2. A protection network of five guards protecting two code segments.

2.2. Oblivious Hashing

The key point of Hoi Chang's protection guards network is to compute the hash value of code segments and react to it. The hash value can be used as part of arithmetical operation or for other variable address evaluation. However, there are two fundamental drawbacks of this scheme: 1) It's easy to detect the operation of reading from the code segment, which means algorithm stealth feature is not good; 2) Attackers can modify the running data by using debuggers to impact the program, which means this scheme cannot resist dynamic tampering.

Therefore, another scheme called oblivious hashing comes out. Since this kind of hash value is computed in the normal function process of the program, it tends to be ignored by the attackers. To implement this scheme, the hash value should be computed by using the original instructions in the source code or in the memory, as shown in Figure 3.

Y. Chen [3] proposes a method to directly insert hash computation code in the source code. The code compute hash values by evaluating variables and control-flow outputs. Related code is inserted in the program to compute the trace hash instead of recording it to detect variable change and control-flow execution. We can change the number of hash evaluation functions to balance performance overhead and protection intensity.

Mattias Jacob and Mariusz H.Jakubowski [4,5] propose another implementation method. As we know, x86 architecture uses fixed-length instructions and there is no need of alignment. This feature can be used to overlap instructions. This method uses overlapped instructions of different basic blocks to realize tamper-resistance. When one basic block is executed, the hash value of another basic block is computed in this process. The advantage of this method is that we don't have to read the code explicitly, so attackers cannot use memory dump to bypass it. The disadvantage is that we can't predict the promptness of the detecting code.

The key point of oblivious hashing is that we try to check if the data value is in the normal range while the program is running.

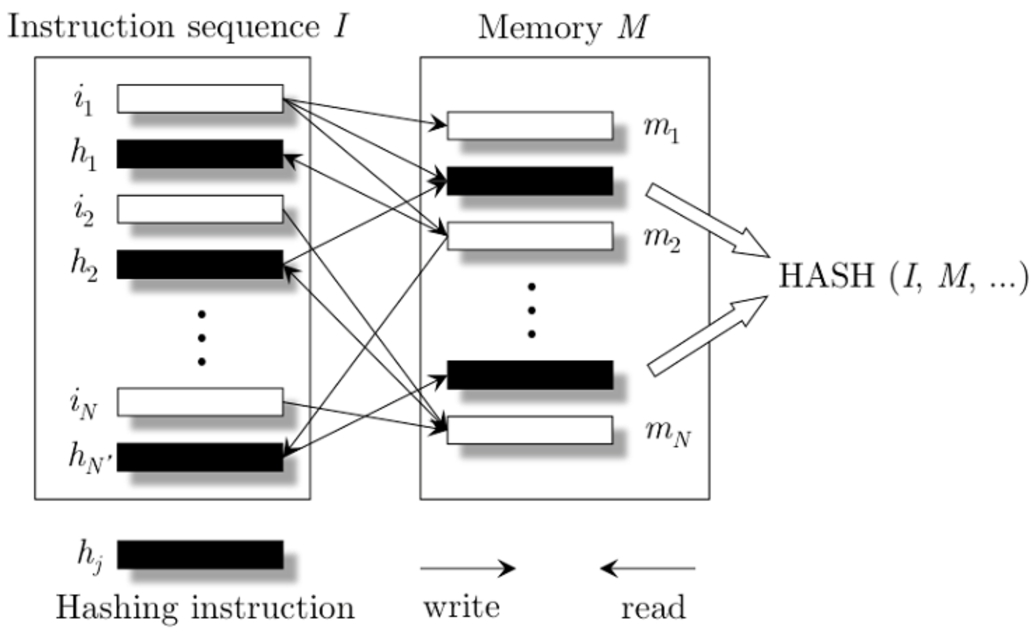


Fig.3. Compute hash value using instructions in the source code or the memory.

2.3. Remote tamper-resistance

The principle of remote tamper-resistance is to separate clients and servers. The program C to be protected is running under a client host, which is controlled by the untrustworthy attackers, while there is another trustful program S on the server host. Program C must keep communication with Program S to be correctly running. Besides providing normal service to the client, the server should also check and respond to the possible tampering behavior on the client host.

Xiangyu Zhang [6] divides one program into two parts, one is the public parts running in the client the other is the hidden parts running in the server. This algorithm only considers about scalar data in the server and make sure client and server are in the same local area network, so that the problem of delay and bandwidth can be solved.

Christian Collberg [7] combines anti-analysis and tamper-resistance by using self-modifying code on the implementation. As shown in Figure 4, the code in client program can change rapidly, which makes it hard to be analyzed. In the ideal situation, the attacker will find the code in the client program changes so fast that before he completes analyzing the current version of program and modify it, the server will send another new program. This method is only in theory and has not been implemented on known systems.

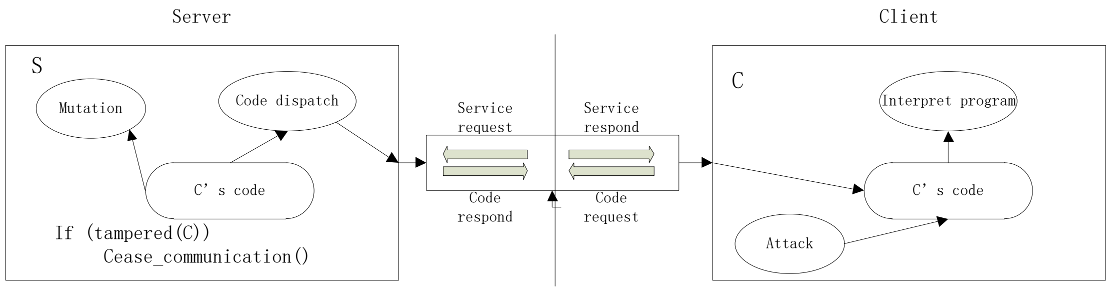


Fig.4. Christian Collberg's remote tamper-resistance method.

2.4. Monitoring running environment

Attackers has developed different strategies to bypass check-respond tamper-resistance. There are two methods: 1) try to find the check code and respond code and delete them. 2) attack from the outside by changing program execution environment.

Glen Wurster [8] proposes a hash attacking algorithm. In the tamper-resistance algorithm based on hash evaluation, two ways will access the code, one is to be executed as instructions, the other is to be used as data to compute hash value. When reading code as data, we should use unmodified code to make sure the hash value doesn't change. To execute the code, we should use the modified code. This method maps on virtual memory address to two different physical address. If the code is used as instructions, it will map to one address. If the code is used as data, it will map to another address. However, there is also a countermeasure to attack this method by modifying the system kernel. Specifically, attackers will firstly copy program P to $P_{\mathbf{orig}}$ and then modify P to the cracked version P'. Finally, the attackers will modify the system kernel to make the host read data from $P_{\mathbf{orig}}$ and read instructions from P'. This attack method relies on the processor type and the memory manage module of operating system.

To deal with this attack, Jonathon Giffin [9] propose a method by adding new code to the program to check whether the program is running in a trustworthy environment. He uses self-modifying code to realize this method. If the program detects itself running in a system that will modify data without impacting code, it will call the respond function to resist tamper.

3. Discussion

3.1. Summary

We can summarize all these implementations of tamper-resistance into five methods:

1) Add new code into the program, to check if the code is modified. Hoi Chang's protection guards network uses this method.

2) Add new code into the program, to check if the program is running in the trustworthy environment. Jonathon Giffin's monitoring running environment uses this method.

3) Check if the data value is in the normal range while the program is running and the control-flow is in a reasonable path. Oblivious hashing uses this method.

4) Remote tamper-resistance: divide the program into two parts, one of which is running in a server or a tamper-resistance hardware to avoid being modified. Xiangyu Zhang's function separating uses this method.

5) Use code obfuscation techniques to make it hard for attackers to understand the program, not to mention modify it. Christian Collberg's self-modifying code uses this method.

3.2. Evaluation System

In software tamper-resistance research area, there is not a widely-accepted evaluation system. In this paper, we refer to the definition of code obfuscation evaluation by Christian Collberg [10] and establish am evaluation system applicable to tamper-resistance. We try to evaluate a tamper-resistance method based on qualitative analysis of protection intensity, overhead and stealth.

- Protection intensity: how effective a tamper-resistance is. It can be defined as whether the program can resist static tampering and dynamic tampering.
- Overhead: space overhead and performance overhead. The former includes the size of added code and other demands of hardware and environment. The latter is mainly about the running time difference after adding protection code.
- Stealth: whether this algorithm can easily be detected and bypassed by attackers.

Table.1. Comparison of different tamper-resistance methods.

	protection intensity		**overhead**		**stealth**
	static tampering	dynamic tampering	space overhead	performance overhead	
Protection guards network	√	×	small	average	bad
Oblivious Hashing	√	√	small	high	good
Remote tamper-resistance	√	√	large	high	bad
Monitoring running environment	√	√	small	average	good

Hoi Chang's protection guards network has a good feature of customizable safety intensity, which is realized by adjusting numbers of protection guards and complexity of defending graph. On the other side, it's not easy to precisely compute the performance overhead of this algorithm because it is related to various features like added code size, location and calling frequency. For example, if we randomly insert the check code into the program and it ends up in a loop function which can be called frequently, the overhead will become high. Thus, when evaluating this kind of algorithm, we should set up the evaluation conditions. As for space overhead, since check code has small size, which is 62 bytes in Hoi Chang's method, it's not a problem. Hoi Chang's method doesn't consider about the stealth problem. The guards code can be easily detected and located then modified, even though complex protection network can be established.

Oblivious hashing improves stealth compared with protection guards network. For example, Y. Chen's method makes it hard for attackers to find the behavior of computing hash value. Oblivious hashing can resist most static analysis, and some kinds of dynamic analysis like memory dump as well. In terms of overhead, because of the execution time relying on computing hash value of overlapped programs, it will be higher. Jacob computes the space and performance overhead of this algorithm and it turns out that the protected program is three times slower than the original one.

In theory, remote tamper-resistance is excellent for dynamic tampering. However, Xiangyu Zhang's function splitting method requires high capacity of computation in the server, as well as network runtime bandwidth. This method is at high cost. As for the overhead, if hardware condition is bad, there will be large delay problem. The overhead depends on the amount of code in the server.

Monitoring runtime environment is the application of self-modifying code. It will function well in dynamic tampering attacks based on memory dump. In the worst situation, this method will cause the overhead like a light-weight system call.

As shown in Table 1, we compare four different software tamper-resistance method in a qualitative way. We can see when the execution efficiency is in higher demand, we should use protection guards network which will sacrifice some protection intensity. When we need very high protection intensity, we should use the other three methods. Under normal conditions, the overhead will increase when the protection intensity becomes higher. In practice, we should fully consider the situation and make balance of intensity and overhead.

4. Conclusion

In the above three sections, we introduce the mainstream software tamper-resistance techniques and find out all of them fail to break through the check-respond mechanism framework. However, respond part is always the weak point of tamper-resistance mechanism, which is often used by attackers as a breach. In existing tamper-resistance research, there is little work about improving respond mechanism. How to effectively combine check and respond parts is a topic worth digging through.

Another trend is to combine tamper-resistance with other software protection techniques. For example, if we combine tamper-resistance with code obfuscation, we can improve the stealth of check-respond mechanism. If we combine tamper-resistance with software watermarking, we can verify the integrity by checking watermarking. Tamper-resistance with software diversity is also a hot topic. In general, combining tamper-resistance with other protection techniques can increase protection intensity and has become a research hotspot.

Acknowledgement

The work was partially supported by the National Natural Science Foundation of China under Grant No. 61472437.

References

[1] Cappaert J. Code Obfuscation Techniques for Software Protection[J]. 2012.

[2] Chang H, Atallah M J. Protecting Software Code by Guards.[C]// CCS 2001, Proceedings of the, ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, Usa, November. 2001:160-175.

[3] Chen Y, Venkatesan R, Cary M, et al. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive[C]// Information Hiding, International Workshop, Ih 2002, Noordwijkerhout, the Netherlands, October 7-9, 2002, Revised Papers. 2002:400-414.

[4] Jacob M, Jakubowski M H, Venkatesan R. Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings[C]// The Workshop on Multimedia & Security. 2007:129-140. [5]Mariusz H.Jakubowski, Protecting digital goods using oblivious checking

[6] Zhang X, Gupta R. Hiding Program Slices for Software Security[C]// International Symposium on Code Generation and Optimization. 2003:325.

[7] Collberg C, Nagra J, Snavely W. bianlian: Remote Tamper-Resistance with Continuous Replacement[R]. Technical Report TR08-03, Department of Computer Science, University of Arizona, 2008.

[8] Wurster G, van Oorschot P C, Somayaji A. A generic attack on checksumming-based software tamper resistance[C]//2005 IEEE Symposium on Security and Privacy (S&P'05). IEEE, 2005: 127-138.

[9] Giffin J T, Christodorescu M, Kruger L. Strengthening software self-checksumming via self-modifying code[C]//21st Annual Computer Security Applications Conference (ACSAC'05). IEEE, 2005: 10 pp.-32.

[10] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations[R]. Department of Computer Science, The University of Auckland, New Zealand, 1997.