Checking Primitives with Guards

Dong Zhang
BSCH
Department of Computer Science
University of Auckland

October 2005

Abstract

The concept of *guards* were introduced by Hoi Chang and Mikhail Atallah as a tamper resistance mechanism. Part of this scheme is a checking primitive—code checksum, which calculates the checksum of a protected code segment. Its counterparts include code hashing [BHT01] and oblivious hashing [YCJ02]. For these two primitives though, there lacks an investigation of constructing a protection system like *guards*. This paper will try to look at the possible approaches and difficulties of constructing such frameworks. We would first briefly describe the checksum guards, then show how code hashing can come in as guards' primitive; and finally we try to discuss the potential problems of an Oblivious Hashing—*guard* scheme.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction and overview **1**

2 Static Primitives with *Guards* **3**
2.1 Code Checksum and Hashing 3
2.1.1 Checksum 3
2.1.2 Code Hashing 5
2.2 Guards 5
2.2.1 Guard with Code Hashing 5
2.2.2 Guard network 6
2.2.3 An attack 7

3 Oblivious Hashing with *Guards* **8**
3.1 Oblivious Hashing 8
3.2 Working with Guards 9
3.2.1 Oblivious Hashing as a Primitive 9
3.2.2 Guard Network and Cross Checking 11
3.2.3 Repairing 12

4 Conclusion and Further Work **13**

Acknowledgements

I would like to thank Prof. Clark Thomborson for the most valuable advices in constructing this term paper.

Also, I would like to acknowledge the CS725 S2 05 course which has offered a truly enjoyable learning experience.

Chapter 1

Introduction and overview

Protection on software against unauthorized modification has gained more awareness over the past few years. Such a protection, more specifically, is expected to stop anyone from tampering the execution code. This concept was formally discussed in Aucsmith's paper [Auc96], where tamper resistance software was defined as "...software which is resistant to observation and modification". To achieve this target, 3 principle approaches have been developed[CT02]. One of them is checking the software execution code, which means the software should be able to verify its own code integrity and take certain actions once an unauthorized tampering is evident. It is believed that the whole mechanism for this approach should consist of at least a detection part and an action part[CT02]. The detection part should be able to recognize modifications on the code and trigger the action part.

The detection step can be farther divided into two steps(Figure 1.1), which are:
1. Obtain a signature representing the to-be-checked code.
2. verify the signature to see if the program is identical to the original one.
For the first step, researchers have designed different primitives that can be used as representations of code segments. For example, Horne et al.[BHT01] suggested using *testers* which computes a hash value of a specific code section; and in Hoi Chang and Mikhail Atallah's [CA01], their *guards* were based on code checksum. On the other hand, a relatively new method called Oblivious Hashing(**OH**), has been proposed by Chen et al.[YCJ02]. This verification primitive, is believed to be able to dynamically capture the pro-

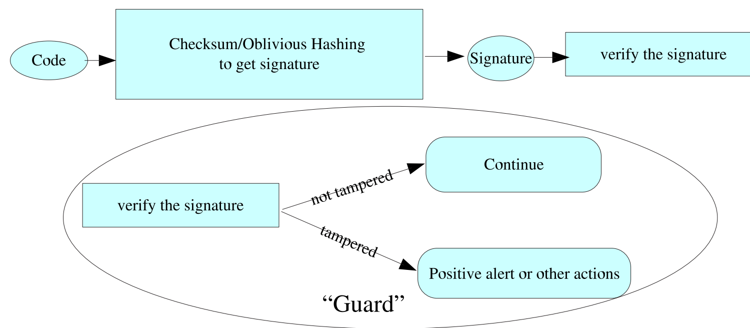

Figure 1.1: A tamper proofing system

gram's execution trace; but in contrast with *guards*, the inventors did not put their emphasis on constructing a complete tamper resistance framework using oblivious hashing.

This paper intends to discuss the issues when code checksum is replaced by two other methods. The paper is organized as following: Chapter 2 will briefly introduce the code checksum used by Chang and Atallah, then present a code hashing guards idea. Chapter 3 will be centered around a comparison between Oblivious hashing and two other primitives. Part of this chapter addresses the issues of a **OH**–guard framework. Chapter 4 will have a conclusion and a preview of possible future work.

Chapter 2

Static Primitives with *Guards*

Static detection means the actual execution code is checked for integrity. Normal approach would first obtain a signature(static primitive) of a code segment, in the form of either a checksum or a hash value. This section will explore how the above two work with guards.

2.1 Code Checksum and Hashing

2.1.1 Checksum

As proposed by Chang et al.[CA01], the guards "Checksum another piece of program code at runtime and verify its integrity". This paper will use the approach implemented in this publication as the way checksum is obtained. We firstly define two terms for clarity.

- target code: code segment that is protected.
- watching code: code that segment that calculates the checksum of target code.

The watching code would firstly set a register for storing the checksum. Then it loads in the memory location of the first instruction in the target code segment; after obtaining the memory content at this location, the watching code

adds the instruction content as binary data into the register and goes for the next instruction of the target code. Therefore, by the end of all iterations, the register should have the sum of all instructions. Once a checksum of the target code is obtained, we can apply a verification on the checksum to see if the code has been altered. An illustration of a simple implementation by Chang et al. is shown as below(Figure 2.1).

```
guard:
      add     ebp, -checksum
      mov     eax, client_addr
for:
      cmp     eax, client_end
      jg      end
      mov     ebx, dword[eax]
      add     ebp, ebx
      add     eax, 4
      jmp     for
end:
```

Figure 2.1: guard Example from [CA01]

Several facts of the this process have made code checksum a promising primitive for tamper detection. In the above described procedure, the signature(checksum) is only based on target code instructions, which means the calculation of checksum does not depend on the runtime environment. So inserting the summing computation can be made simple. As argued in [CA01], this checksum calculation routine can be inserted into binary code after compilation as long as it does not corrupt the memory space used by normal code. Therefore, it enables a third party to apply protection on a compiled software product.

On the other hand, it also has some defects . For example, reading a code segment is considered as atypical behavior, which can lead attackers to locate watching code fairly soon. And also since the checksum only depends on the static shape of target code segment, attackers can patch the return checksum value at runtime to pass the watching code. [YCJ02] Further more, with careful modification, one can change or patch the target code to make its checksum remain same while the actual instructions may be changed for malicious purpose.

2.1.2 Code Hashing

Other than checksumming the target code, hashing can also be used to obtain a signature for the code being protected. Instead of simply summing the instructions, Horne et al. [BHT01] let the watching code compute a hash value based on "a large interval of the executable (several hundred kilobytes)"; then according to their argument, the chance that a changed interval mapping to the correct hash value can be reduced to the level of 2^{-32}, if "a good choice of hash function" is selected.

Despite many of the strengths listed in their paper, code hashing method also has the similar problems as in code checksum. For example, it also needs to read its own code segment, which stands out other normal operations; it is also vulnerable to runtime patching attack, once the correct pre-stored hash value is figured out.

Both code checksum and hashing obtain signatures from static code shapes. Next section is devoted to explaining the whole guarding framework.

2.2 Guards

The guarding framework proposed by Chang and Atallah "is provided by a network of execution units (or guards) embedded within a program" [CA01]. These guards are designed to be capable of doing any computations [vO03]. We would like to spend this section giving a brief design of a hashing–guards framework.

2.2.1 Guard with Code Hashing

The idea of letting Guard work with code hashing is to replace the checksum computation with a linear hash function illustrated in [BHT01]. Take the sample checksum guard in (Figure 2.1) for example, instead of adding the content at memory address **eax** to register **ebp**, we can hash it with **ebp** then update this register with the new hash value. A simple hash–guard version of the previous example 2.1 can be shown as in (Figure 2.2).

```
guard:
        add ebp, -hash
        mov eax, client_addr
for:
        cmp eax, client_end
        jg end
        mov ebx, dword[eax]
        xor ebp, ebx
        add eax, 4
        jmp for
end:
```

Figure 2.2: A hash checking guard

More than just detection, guards' repairing function can restore a modified code segment to its original. This function is considered as a possible action part. The way it was implemented in [CA01] was when watching code detects a modification, a clean version is used to overwrite the damaged target code. This fixing mechanism gives programs a choice to run as if unmodified. But a sequential constraint is enforced here, that guards should get run before target code does. This is one of the major concerns when designing a guard network.

2.2.2 Guard network

Similarly, the guard network suggested by Chang et al.[CA01] can be constructed in hashing–guard framework. The idea of guard network enables guards to verify other guards , which means watching code becomes target code. Thus an attempt to poison a guard will trigger other guards to sound alert. Once a guard cycle is formed, an attacker has to disable all the guards on this cycle at the same time to get rid of protection. One sample guard network used in [CA01] is shown as below in Figure 2.3.

There is a requirement when constructing the guard network. As damaged code needs to be fixed before being executed, repairing action should dominant the target code in the control flow sequence. This can lead to some

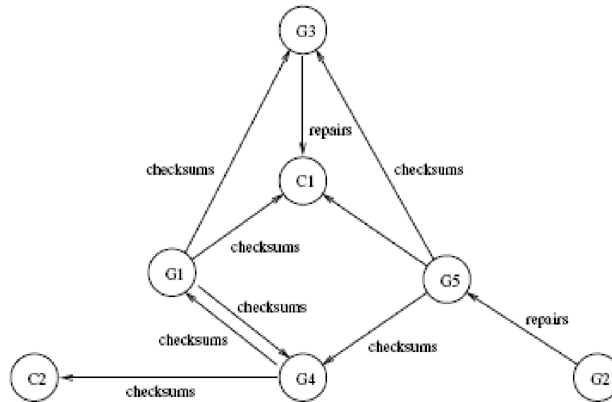

Figure 2.3: A guard network from [CA01]

constraints on selection of guard placement. Such constraints would pose a significant difficulty in the Oblivious Hashing–Guard framework. Later section will have more about this.

2.2.3 An attack

A generic attacking model developed by Wurster et al.[ea05] exploited the fact that many modern microprocessors failed to translate both the code and data virtual memory address into the same set of bytes. Thus it is possible that "when running, the processor would execute the attackers modified instructions; when checksumming, the application would read a copy of its unmodified code."[ea05] Processors like UltraSparc, x86 have been proved to be vulnerable to such attacks. This also justified the arguments in [YCJ02], the atypical behavior that a program reads its own code can let the guard be found out fairly soon.

Chapter 3

Oblivious Hashing with *Guards*

3.1 Oblivious Hashing

Oblivious Hashing was a concept proposed by Chen et al. in [YCJ02] as a software verification primitive. Unlike the previous static tamper detection, this method tries to find a signature of a piece of code based on its execution behavior. The way this is achieved is through capturing the memory content at runtime. Detailed explanation can be found in Chen's paper.

Although not clearly mentioned in their writing, the basic assumption that makes it work is the input and output can precisely define an instruction. The authors argued that with careful analysis, it is possible for an attacker to modify the code and still produce the correct oblivious hash. If this is the case, the modified code would make the same impact to watched memory spaces as the original program does. As we shrink down the protection region, the above attack become significantly hard to achieve because the modification to a small set of instructions is very much constrained by expected input/output values pair. On the other hand, we believe potential vulnerabilities do exist, such as memory patching attack. Chen's paper has more discussion about these issues.

The property of relying on memory content changes has substantially distinguished oblivious hashing from code checksum and hashing. This feature takes **OH** beyond some of the limitations that the other two primitives have.

For example, reading memory is a typical operation which does not easily attract hackers' attention. So even without obfuscation techniques, Oblivious Hashing fairly suffice the requirement for stealthiness. Another advantage is that unlike in static detection, oblivious hash is dependent on input and output, and obtained at runtime. Therefore, the attacking model mentioned in Chapter 2 loses its base that a damaged code is run but its original copy is checked.[ea05].

3.2 Working with Guards

Assuming in a hardware environment as described in [ea05], where in virtual memory, the logical boundary between executable code and data can be identified, we try to address the differences between using oblivious hash and checksum. Meanwhile other issues will also be investigated.

3.2.1 Oblivious Hashing as a Primitive

Different from checking code checksum, the guards in this case need to read out the generated hash values from runtime data memory and compare with a pre-stored value, whereas checksum need only to operate on the executables' side of the memory space.

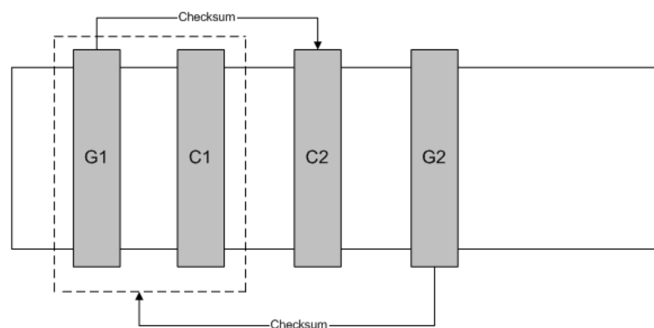

Figure 3.1: Guards with checksum after [CA01]

Figure 3.1 follows the design idea in [CA01]. The guarding agents G1 and G2 calculate a code checksum of target segment and do a comparison with

the checksum of original code. The situation in oblivious hashing, however, would look like in Figure 3.2.

In the above graph, the set of hashing instructions H1 is used to protect nor-

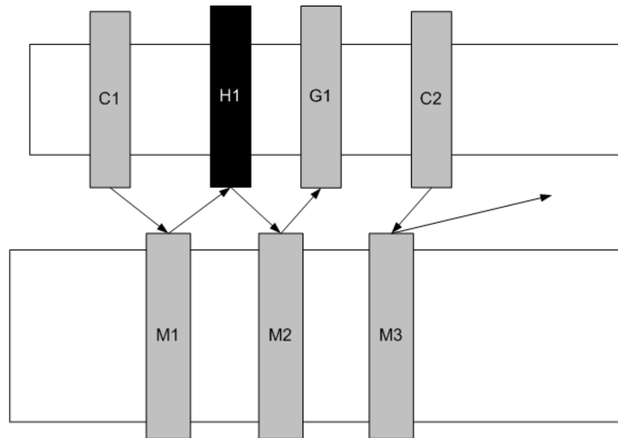

Figure 3.2: Guards with oblivious hash

mal code C1. It would generate a hash value based on the memory content M1 that contains C1's output information. Then H1 saves the hash value into another memory space M2, where guard G1 can pick up and do the checking. Unlike in Figure 3.1, where the checksum calculation is combined with checking part into a guard, we would like to separate signature (**OH**) acquisition out from the checking routine. This would leave the guard with the only functionality of comparison. The reason to employ such a redesign is that we can possibly insert some normal executions between H1 and G1. Since all of the C1, H1 and G1 behave as normal instructions that read/write data from/into runtime data memory, a delay in guarding action would make the protection looks stealthier.

Following the above topic, it is probably a good idea to reuse memory space as much as possible, especially where resources are limited. Also, when protection is to be applied on a compiled product, how long M1 retains the information generated from C1 must be studied. In any case, in order to give C1 coverage, we need to make sure that M1 is not overwritten by other codes before we obtain the hash. this factual constrain may limit the delay we can put between C1 and H1, but since we are in control of both H1 and G1 and their memory usage, such constrain should be much looser between

H1 and G1.

3.2.2 Guard Network and Cross Checking

In [CA01], a guard network was constructed to strengthen resilience against attacks. More specifically, a chained guard cycle mentioned in previous section would pose a huge difficulty to hackers. In the smallest case, such a cycle contains two guards, which can also be named cross checking.

Cross checking an oblivious hash would have more issues. The reason that cross checking can be implemented with checksum is the guard code is static while oblivious hashes depend on runtime execution. Abstractly speaking, suppose we have two oblivious hashing guards G1 and G2 trying to cross check each other. In order to verify that G2 has not been modified, G1 has to check the oblivious hash generated by G2. And this has to happen at the time when execution of G2 should have terminated. On other hand, to let G2 verify G1's hash, G2 has to wait until the hash of G1 is computed. To resolve this sequence conflict, we may choose to let two guards cross check each other part by part. see (Figure 3.3)
The above scheme would inevitably construct an infinite recursive checking.

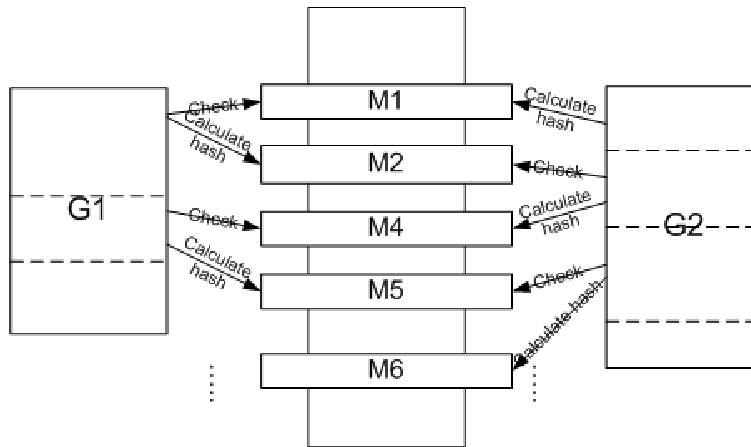

Figure 3.3: oblivious hashing guards cross check each other

As also argued by Chen et al., an implicit way is needed to terminate recursion without highlight the positions of guards to attackers. In addition, at

the point of forced stop, it is possible that certain parts of the guards are left outside the protection coverage. Say as in the above example, if we stops at checking M5, the third section of G2 will be exposed to be unprotected; and moreover, if such an intention is recognized by an attacker, it may be also possible that all the previous checkings are tracked out.

3.2.3 Repairing

One of the functionalities implemented in code checksum guard is the ability to repair a piece of damaged code. Section 2 and the paper [CA01] described how it works. The basic requirement for this feature is that a modified code segment must be checked before it is run, then there is chance to repair the damage ahead of execution. Since oblivious hashing captures runtime behavior, it is not possible to obtain a hash value until the protected code is actually run. This conflict poses a major difficulty in adding repairing mechanism into oblivious hashing guards.

Continuing with the topic of clean code restoration, an interesting question could be raised at higher security requirement level, namely—what if a piece of modified code will already have caused harm by the time we detect it. This is a discussion about the requirements to a complete tamper resistance system. Whether there is a novel way that oblivious hashing–guards framework can address this problem is still to be seen.

Chapter 4

Conclusion and Further Work

To construct a guarding framework with different primitives require different considerations. A tradeoff between static and dynamic signature acquisitions has been the theme of this paper. Although oblivious hashing is defensive to the attacking model that can be applied to static detections, it also faces many issues when guard framework is to be produced.

As tamper resistance is an active research area, an attractive topic could be looking for a way to effectively combine different types of techniques, to produce a stronger protection system than individuals.

Bibliography

[Auc96] David Aucsmith. Tamper resistant software: An implementation. *Lecture Notes In Computer Science*, 1174:317–333, 1996.

[BHT01] Casey Sheehan Bill Horne, Lesley Matheson and Robert E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. *Lecture Notes In Computer Science*, 2320, 2001.

[CA01] Hoi Chang and Mikhail Atallah. Protecting software code by guards. *Lecture Notes In Computer Science*, 2320:160–175, 2001.

[CT02] C.S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation- tools for software protection. *IEEE Trans. Software Engineering*, 28, June 2002.

[ea05] Glenn Wurster el al. A generic attack on checksumming-based software tamper resistance. *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

[vO03] P.C. van Oorschot. Revisiting software protection. *In Proc. of 6th International Information Security Conference (ISC 2003).Lecture Notes In Computer Science*, 2851, October 2003.

[YCJ02] Matthew Cary Ruoming Pang Saurabh Sinha Yuqun Chen, Ramarathnam Venkatesan and Mariusz H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. *Lecture Notes In Computer Science*, 2578:400–414, 2002.