Protecting Software Codes By Guards

Hoi Chang, Mikhail J. Atallah
Center for Education and Research in
Information Assurance and Security
&
Arxan Technologies

Protecting Software Code By Guards [*]

Hoi Chang and Mikhail J. Atallah

CERIAS, Purdue University
and
Arxan Technologies, Inc.
{changh,mja}@cerias.purdue.edu

Abstract

Protection of software code against illegitimate modifications by its users is a pressing issue to many software developers. Many software-based mechanisms for protecting program code are too weak (e.g., they have single points of failure) or too expensive to apply (e.g., they incur heavy runtime performance penalty to the protected programs). In this paper, we present and explore a methodology that we believe can protect program integrity in a more tamper-resilient and flexible manner. Our approach is 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 (checksumming the program code is one example) and a network of them can reinforce the protection of each other by creating mutual-protection. We have implemented a system for automating the process of installing guards into Win32 executables. [1] Experimental results show that memory space and runtime performance impacts incurred by guards can be kept very low (as explained later in the paper).

1 Introduction

Software cracking is a serious threat to many in the software industry. It is the problem in which a *cracker*, having obtained a copy of the software he wants to attack, succeeds in breaking the protection that comes built into it. Typically, crackers would create modified versions of the software, or *crackz*, whose copy protection or usage control mechanisms have been disabled. Cracked software can then be illegally redistributed to the public, exacerbating the software piracy problem. With commerce and distribution of copyrighted multi-media rapidly moving online, the need for software protection is even more urgent than before: client software code running on untrusted machines has to be secured against tampering.

[*] Parts of this research were supported by CERIAS and the Purdue Research Foundation.

[1] A US patent on the technology has been filed by Purdue University and licensed to Arxan.

What makes software cracking so widespread is in part caused by the simplicity of direct inspection and modification of binary program code with existing software debugging and editing tools. Here is an example of how a program requiring online registration can typically be cracked. The program would normally go through a long sequence of procedures asking for a registration serial number from its user, and then in a stealthy manner, comparing a function of the true serial number with the same function of the entered one. After comparing these two items, however, the program then ends up deciding the authenticity of the software user with one single instruction, typically a conditional branch that decides whether the software can henceforth be used. To defeat the entire registration scheme, one only needs to replace that single instruction in the binary file with an unconditional jump (that jumps to a desired location), or by a sequence of smaller no-ops (that do nothing except letting the execution flow to the desired location naturally). The problem with this protection scheme is that the branch instruction is a single point of failure. With sophisticated program debuggers and hex editors (such as SoftICE [8] and HIEW), attackers are able to trace targeted parts of the program, pinpoint the code they need to compromise, and finally apply changes to the program files.

Many commercial protection schemes employ what we call monolithic protection schemes, in which protection is enforced by a single code module in the program but which is loosely attached to the program and thus can be disengaged easily (using methods similar to the above example).

How can software be perfectly secured against cracking? This looks like an impossible task *if* one interprets "cracking" as "eventually cracking", i.e., after a long time. The fact that crackers have huge cracking resources makes successful attacks possible after a long enough time (because they could rewrite the software from scratch after sufficient analyses of the code). However, it is possible to "raise the bar" for attackers and make it *sufficiently secure*. Because many software developers only hope for a minimum length of time during which they could sell a large enough number of a newly released product, securing software code until the end of the period is cost-effective.

Protection mechanisms that can effectively protect software running in untrusted environments should have the following properties:

- **Resilience**: The protection has no single points of failure and is hard to disable.
- **Self-defense**: Able to detect and take actions against tampering (i.e., code modification).
- **Configurability**: Protection is customizable and can be made as strong as one needs.
- **White-box security**: Because any scheme for protection is likely to become publicly known over time, its strength should not be based on its secrecy but rather on the knowledge of a secret key used at protection-install time (but not stored anywhere within the protected program).

This paper describes a security framework and system (having the above desirable properties) for protecting program code against tampering. We extend the traditional ideas of having code check and modify itself to a general setting, in which a program is protected by a multitude of such functional units (called guards) integrated with the program. To defend themselves against attacks, guards form a network by which they protect each other in an interlocking manner. The network of guards is harder to defeat because security is shared among all the guards, and each of them is potentially guarded by other guards. The fact that there are many ways to form a guards network, makes it hard for attackers to predict its form. Furthermore, more guards can always be added to the program if a greater level of protection is desired.

We believe that this guarding framework can advance the state of software protection by making protection schemes derived from it more sophisticated than existing schemes, and easier to apply. Using our system, we show that protecting programs using this guarding framework is possible. Also, we show that the guarding process can be automated (so that it will become unnecessary for one to go through a laborious and error-prone process of manually guarding the program code).

The paper is organized as follows. Section 2 provides some related work in this field. In section 3, we describe the protection framework and discuss its security issues. In section 4, we introduce the system we built. This is then followed by experimental results in section 5. The final section concludes and describes enhancements to the system that are currently being implemented.

2 Related work

The protection mechanisms for software protection involve two main approaches to the problem: hardware-based protection (which relies on secured hardware devices for protection), and software-based protection (which only relies on software mechanisms for protection).

One hardware solution is the use of secure coprocessors (or processors) [18, 19, 15]. In secure coprocessors, programs or portions of them can be run encrypted, so their code is never revealed in untrusted memory. Thus secure coprocessors can provide the programs isolated execution environments that are difficult to tamper with. Although tamper-resistant, this approach requires the use of special hardware for executing programs, which may not be cost-effective for widespread use (say, in typical home-user environments).

Using smart cards for software protection is another solution [2, 10]. Since smart cards contain both secure storage and processing power (although some only provide secure storage), security-sensitive computations and data can be processed and stored inside the cards. A major difference between smart cards and secure coprocessors is that the former are resource-tight (i.e., limited storage space and processing power), and can be used to protect only small fragments of code and data.

Dongles have long been in use by the industry for software protection. They are the hardware keys plugged in the computer, without which the programs that came with the dongles cannot execute. The major drawback of dongles is that each dongle-enabled software usually requires a different dongle. Moreover, the protection can often be bypassed because the communication traffic between dongles and their programs can be intercepted and modified.

One software-based approach for protection is code obfuscation, which "scrambles up" program code so that it results in some executable code that has the same functionality as the original but is difficult to understand and analyze [11, 5, 6, 4, 12, 13, 7, 17]. This form of protection is more flexible than the hardware-based one because it does not require special execution environments. But exactly how secure it is is still a matter of debate [3].

There are other software-based approaches to the problem as well. These include the use of self-modifying code [9] (code that generates other code at run-time) and code encryption and decryption [14] (partially encrypted code self-decrypting at run-time). A hybrid approach of the above has been proposed by Aucsmith [1], which involves the use of cryptographic means to decrypt and encrypt a window of security-sensitive program instructions before and after each execution round of those instructions. One of the problems with this approach is that it does not scale well as the size of the aboive-mentioned "window" gets large (because of the time taken by encryption and decryption).

3 The guarding framework

In this section, we describe our guarding framework and explore some of its security issues on an informal basis.

3.1 Guards

In our guarding framework, protection is provided by a network of execution units (or *guards*) embedded within a program. Each guard is a piece of code responsible for performing certain security-related actions during program execution. Guards can be programmed to do any computations, and the following are two useful ones:

- **Checksum code** [2]: Checksum another piece of program code at runtime and verify its integrity (i.e., check if it has been tampered with). If the guarded code is found altered, the guard will trigger whichever sequence of actions is desired for the situation, ranging from the mildest of silently logging the detection event, to the extreme of making the software unusable (e.g., by halting its execution, or better yet, causing an eventual crash that will be hard to trace back to the guard). If no code changes are detected, the program execution proceeds normally. Programs guarded by checksumming guards are made, in some sense, "self-aware" of their own integrity.

- **Repair code**: Restore a piece of damaged code to its original form before it is executed or used (as data). One way to achieve code repairing is to overwrite tampered code with a clean copy of it stored elsewhere. This repairing action effectively eliminates the changes done to the code by an attacker, and allows the program to run as if unmodified. Repairing guards provide a program with "self-healing" capabilities.

3.2 Guards network

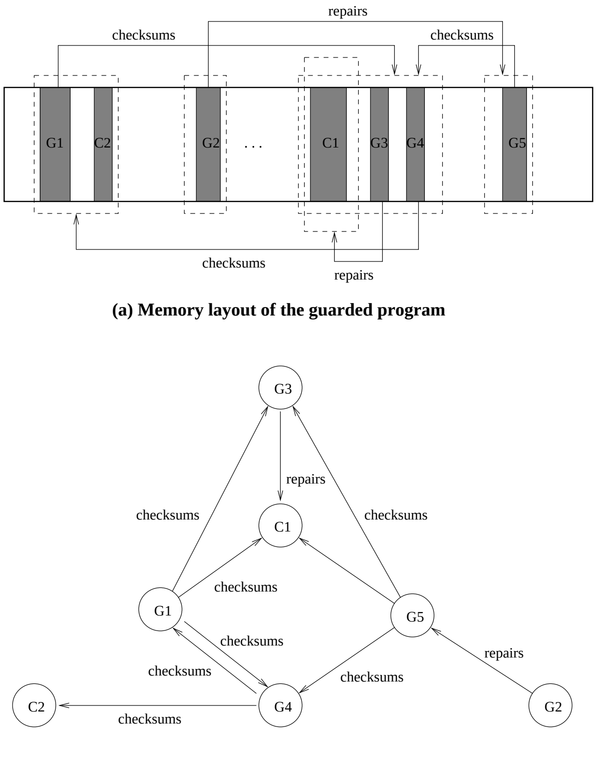
A group of guards can work together and implement a sophisticated protection scheme that

[2] In this paper, "code" refers to both the runtime data and executable code of a program.

is more resilient against attacks than a single guard. For example, if a program has multiple pieces of code whose integrity needs to be protected, then it can deploy multiple checksumming guards for protecting the different pieces. Besides sharing the load of protection, guards have the flexibility to protect one another. Figure 1 shows a possible guarding scenario in which two security-sensitive regions of a program, $C1$ and $C2$, are protected by both checksumming and repairing guards. Figure (a) shows the memory image of the guarded program, in which $C1$ and $C2$ are guarded by guards $G1, \ldots, G5$ in an interlocking manner. The corresponding guarding relationships can be more clearly depicted by a *guard graph* in Figure 1 (b), where $C1$ is repaired by $G3$ before $C1$ executes, and the repaired $C1$ will subsequently be also checksummed by $G1$ and $G5$ (but $G2$ will repair $G5$ before $G5$ executes).

In order to perform their duties, a network of guards need to be placed into the program and hooked to its execution flows in an appropriate way. For example, a repairing guard has to be inserted into a point in the control flow that is to be reached first (in execution order) before the guarded code is reached; i.e., a repairing guard has to *dominate* the target code in their control-flow locations. On the other hand, a checksumming guard must be installed at a point at whose execution time the code to be checksummed must be present in the program image. Figure 2 (a) shows a graph that depicts the dominance relationships between different pairs of the nodes in Figure 1 (e.g., $G3 \rightarrow G1$ means location of $G3$ dominates that of $G1$).

Figure 2 (b) shows two possible scenarios in which the network of guards can be installed into the control flow graph of a program without violating the partial ordering of their executions specified in (a). As seen from the figure, the larger a program, the more ways there are to deploy the network of guards.



(a) Memory layout of the guarded program

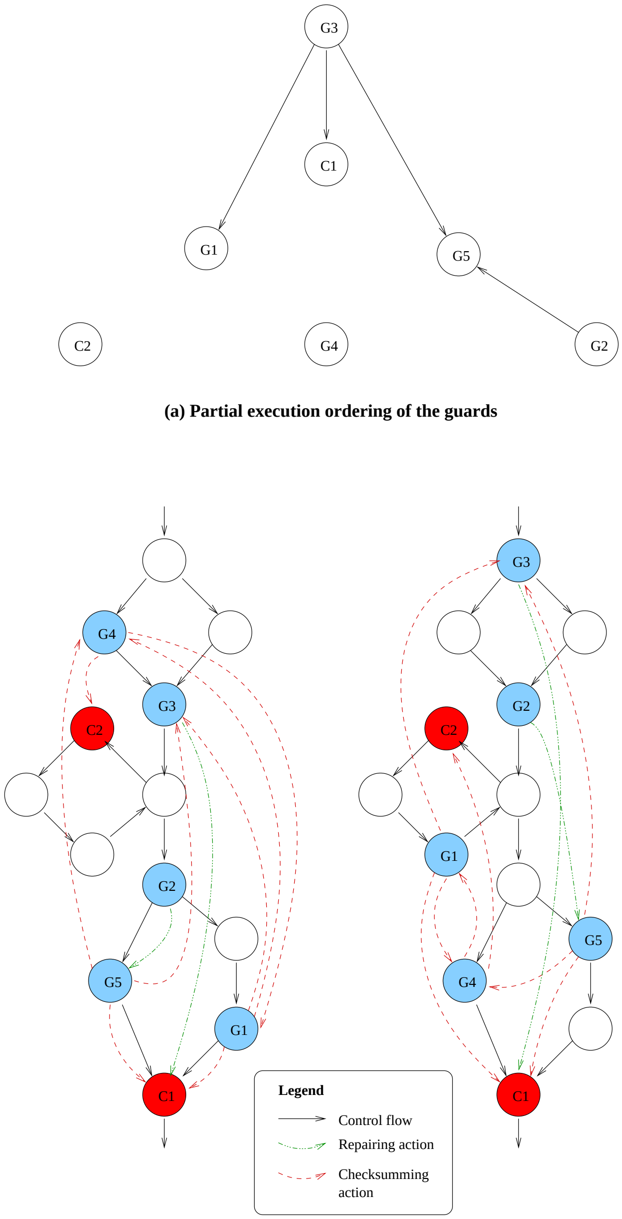
(b) The corresponding guard graph

Figure 1: Program image guarded by five guards and the corresponding guard graph.

3.3 Security

Contrary to monolithic protection schemes in which security is enforced by single security modules, protection by guards enjoys the following advantages:

- **Distributedness**. There is no single point of entry (exit) into (out of) the guards network because its individual com-



(a) Partial execution ordering of the guards

(b) Two possible placements of the guards in a CFG

Figure 2: Guards network installed into a program CFG.

ponents (i.e., guards) are invoked at different points at runtime. This makes it much harder for an attacker to detach the network from the program. To defeat the guards, their locations and guarding relationships need to be identified (an even more difficult task if the program is large and complex).

- **Multiplicity**. Multiple guards can be used to guard a single piece of code, providing it a variety of protection at different times.

- **Dynamism**. There are many ways in which a guards network can be configured. For example, a group of ten guards can form different types of formations, ranging from simple trees to general directed graphs with cycles. Even if one knows the general mechanism for guarding programs, one is still faced with the actual deployment scheme in the program. Furthermore, a fixed formation can be installed in various ways because parameters such as the physical locations of guards and the exact ranges of code they guard could vary from installation to installation. (Consider that each installation is driven by a different random number.)

- **Scalability**. It is easy for the levels of guarding to be scaled up for larger or more security-critical programs by adding to them more guards.

3.3.1 Strengthening the guards network

A guard cycle is a circular chain of guards each of which protects its next neighbor, forming a cycle of guarding relationships in the guard graph. Such a formation allows each guard in the cycle to be protected without any "loose ends" (i.e., unprotected guards). Defeating a guard cycle requires all of the guards to be disabled at the same time. How to implement checksumming in guard cycles is itself an interesting problem, because the checksumming function should have a 1-way property (we have solved the problem but due to page limitation, we omit the discussion in this paper).

The above property of guard cycles leads to a more general guards strengthening scheme: Connect any disconnected components in a guard graph in such a way that each guard in the graph can be reached by the rest of the guards (i.e., the resulting guard graph is strongly connected). As a result, strong connectivity forces the amount of attack efforts to be scaled up proportionally to the total number of the guards deployed in a program.

3.3.2 Strengthening individual guards

The level of difficulty in locating guards and understanding their semantics depend on how "stealthy" and tamper-resistant the guards are.

- **Stealthiness**. Guard code should have no recognizable signatures (e.g., fixed set of instructions) that an attacker can statically scan for. Also, their actions should be made as inconspicuous as possible. For example, instead of instantly sounding an alarm upon detection of an attack, guards should delay such an action until a later time when it is unclear why and how it has taken place. To thwart sophisticated runtime program analyzers from identifying the checksumming or repairing actions of guards, logical boundaries between the executable code and runtime data of a program should be blurred. For example, the code sections are made to contain runtime data, and conversely, the data sections are made to contain executable code.

- **Tamper-resistance**. In situations where the location of a guard has been identified, it is important to have the guard protect itself (besides having other guards protect it). One effective way to achieve this is to obfuscate the guard code. There are many ways to do so. A simple way would be to rearrange its instructions and mix them with dummy code [12]. More aggressive obfuscating transformations are possible and can make the resulting code very difficult to reverse-engineer. Such transformations involve both control and data flow obfuscations. Some particular techniques are discussed in [17, 7, 5, 6].

4 Description of system

We have built Version 1.0 of a system for guarding Win32 executables. It takes an EXE program file as input and inserts into it guards that can perform functions such as checksumming and repairing program code. The guard installation is an automated process guided by a user-provided guarding script that specifies what and how guards will protect the program code and themselves (i.e., the description of a guard graph). Figure 3 gives an overview of our system.

Our system processes binary code directly because high-level code lacks much binary information that guards need (such as memory addresses and binary contents of the program code). Also, manipulating code at the bi-

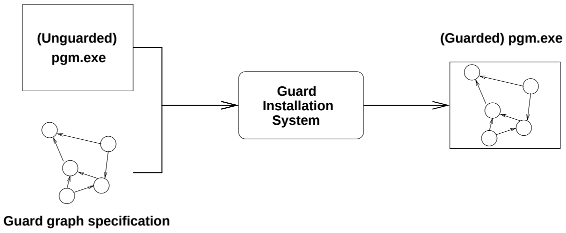


Figure 3: The guarding system

nary level makes it easier to transform program code to whatever form is desired without typical structural restrictions imposed by high-level languages.

Guard installation by our system involves inserting a guard into the program and parameterizing it appropriately. We call this *guard instantiation*, in which guards are instantiated from predefined guard templates, which are object code and stored in a database (of course these are "polymorphic" in the sense that even if two of them have the same functionality they look different; this prevents attacks based on pattern matching techniques). Below is a simple example of a guard template, which is programmed to corrupt stack frame pointer `ebp` if the computed checksum is different from `checksum`. [3]

```
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:
```

During instantiation of the guard, the system initializes `client_addr` and `client_end` with the addresses of the target code range that the guard needs to protect. The other parameter, `checksum`, is later patched to the guard code when the checksum value of the target range has been obtained by the system.

Figure 4 shows the memory image of a program after it has been installed with 307 guards. (Its linear address space is represented by a two-dimensional space for easy interpretation of the image contents.) Shown in dark colors are the four executable regions of the program. (The white regions are file formatting and data areas of the program.) These four regions include three types of code: original (executable) program code, the inserted guard code, and the code protected by the guards (which includes portions of the program code and guard code).

It is important that guard installations be automated. If done manually, it is a very laborious and error-prone process, as it requires one to deal with binary information in the program files directly (consider implementing by hand a function that checksums its own code). The manual task will become more difficult and time-consuming as the number of guards and complexity of their inter-locking relationships increase. Furthermore, programs with "hand-patched" checksumming guards would be very hard to maintain because one cannot change the code without recomputing checksums of the modified code. Our system streamlines the guarding process by separating the task of soft-

[3] The sample template is shown in the NASM assembler language [16].

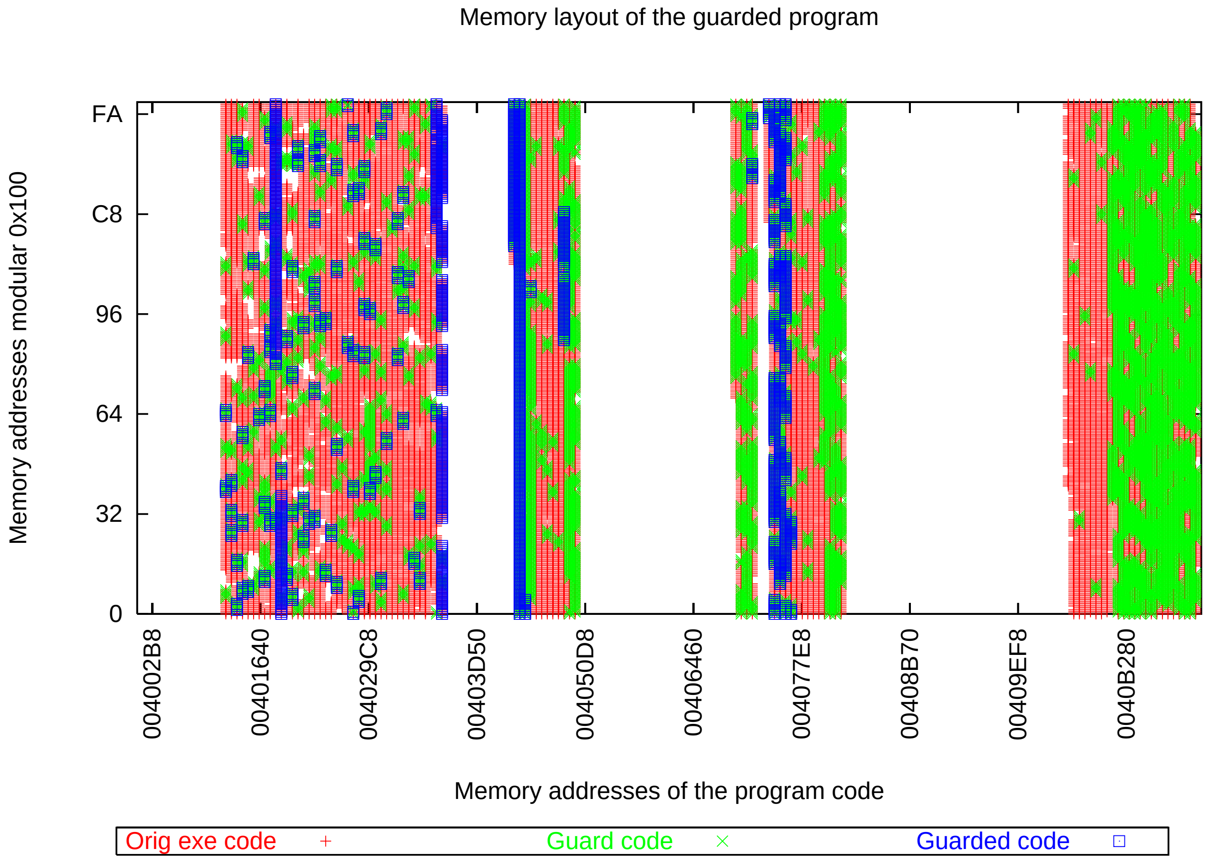


Figure 4: The memory image of a program heavily guarded by 307 guards

	Before guarding		After guarding (without increasing file size)			
	File size	**# instructions**	**File size**	**# instructions**	**# guards installed**	**Avg guard size**
gzip	172 KB	38348	172 KB	38897	25	76 bytes
disasm	376 KB	54931	376 KB	56456	70	75 bytes
avi2mpg	380 KB	51647	380 KB	54913	144	78 bytes

Figure 5: Statistics of the guarded programs and their guards

ware development from that of software protection (which is now done post-compilation).

5 Experimental results

In this section, we examine how much program resouces guards would need from several software applications. By program resources we mean increases in program size and program execution time. We applied our system to three software applications: `disasm`, `gzip`, and `avi2mpg`. `disasm` is an Intel x86 disassembler that is branch-intensive; `gzip` is a GNU file compressing and decompressing tool that has a mixed use of branches and loops; and finally, `avi2mpg` is a Win32 application which converts an AVI video file into an MPEG one. Our experiments were conducted on a Pentium III 600MHz machine running Windows NT.

5.1 Impacts on program size

The amount of program space required for storing guard code is proportional to the number of installed guards and their average size. But sometimes, Win32 executables can accommodate a number of guards without needing more file space. To illustrate this, we ran our system on the test programs and installed into each as many guards (of the same size) as possible (while keeping their file sizes unchanged). Figure 5 shows the maximum numbers of guards that can fit into each program without increasing its size. For the sake of this experiment, the guards inserted into each program were instantiated from the same guard template (of size 62 bytes), which is similar to the one shown previously. The instantiated guards need more bytes because extra instructions are needed to hook their code to the program flows (of course in a "production run" of our system we would use guards having a variety of sizes).

We believe the issue of storage space does not pose a problem to guarding. As storage media such as hard disks are getting more spacious and cheaper, software applications also tend to expand in size (because more functionality can be included). Increasing the size of a program by a few kilobytes (as a result of guarding) does not even show up on the radar screen when compared to the natural increase in the size of software.

5.2 Impacts on program performance

In this section we examine how guarding affects program performance. In particular, we want to answer the basic question: Would guards impose prohibitive time-performance penalty on programs?

We tested the performances of `disasm`, `gzip`, and `avi2mpg` as follows. For each program, its original performance (before guarding) was measured. Then we created a set of guarded versions of each program, each version executing a different number of guards. Inserted into the program at random locations, the guards were invoked *every* time the execution flow reached them. All of the guards performed checksumming on some piece of code of 0x50 bytes long using the same checksumming algorithm. The execution times of this set of guarded programs are keyed as "uncontrolled guard invocations" in Figures 6, 7, and 8.

These performance results (and many others that we ran) show that if guards are placed within highly repetitive loops and execute as many times as they iterate, the performance would suffer. But the results also suggest that if the execution frequency of guards is restricted to a small number, then the programs would likely perform well without much degradation in speed. Indeed, in many cases, guards do not need to execute over and over again if all they do is to repeat the same checksumming or repairing actions that they have repeated many

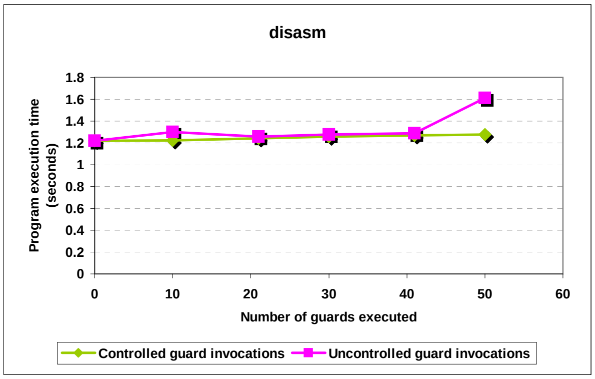


Figure 6: Comparison between the runtime performances of disasm in two scenarios

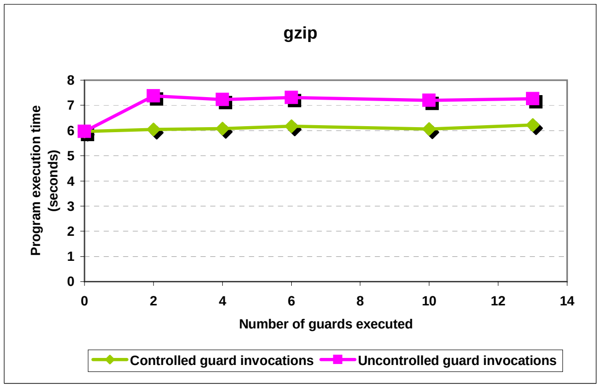


Figure 7: Comparison between the runtime performances of gzip in two scenarios

times already.

To test how controlled invocations of guards affect program performance, for each test program we created another set of guarded versions of it, which were exactly the same as the set created earlier except that in this case each guard executed once only (no matter where it was located in the CFG). The execution times of these guarded programs are shown in the same figures as "controlled guard invocations." Clearly, the new results indicate only slight increase in execution times, as compared to the previous results.

In situations where one could avoid installing guards within performance-sensitive code, the performance results are expected to be better than those reported here. (Our system includes a graphical user interface that makes it easy to highlight portions of program code where guard-installation is recommended, and portions where it is not recommended, in addition to highlighting which portions of the program code should be guarded.) The reason we decided to not use this facility in our experiments is the difficulty in quantifying what "good guard-placement hints" are, and in accounting for their variability from one test application to the next. Instead, we ran our system in "random guard-installation" mode, because it makes comparisons easier between one protected application and another.

6 Conclusion and further remarks

We have explored a software-based methodology for making program code tamper-resilient by using guards. Guards are special code segments in the program which, when deployed collectively, can make the following possible:

- **Distributed protection**. Spreading the load of protection among guards essentially eliminates the "single point of failure" problem.
- **Variety of protection schemes**. There are many ways to group the guards to-

gzip							
Total no. of installed guards		0	5	10	15	20	25
No. (%) of guards executed in a typical run		0 (0%)	2 (40%)	4 (40%)	6 (40%)	10 (50%)	13 (52%)
% increase in exe time	**Uncontrolled guard invoc. (bad)**	0.0%	23.5%	21.2%	22.4%	20.6%	21.7%
	Controlled guard invoc. (preferable)	0.0%	1.1%	1.6%	3.4%	1.4%	4.1%
disasm							
Total no. of installed guards		0	14	28	42	56	70
No. (%) of guards executed in a typical run		0 (0%)	10 (71%)	21 (75%)	30 (71%)	41 (73%)	50 (71%)
% increase in exe time	**Uncontrolled guard invoc. (bad)**	0.0%	6.7%	3.3%	4.8%	5.7%	32.2%
	Controlled guard invoc. (preferable)	0.0%	0.5%	2.1%	3.1%	4.3%	4.9%
avi2mpg							
Total no. of installed guards		0	27	55	82	109	136
No. (%) of guards executed in a typical run		0 (0%)	2 (7%)	4 (7%)	6 (7%)	10 (9%)	12 (9%)
% increase in exe time	**Uncontrolled guard invoc. (bad)**	0.0%	-0.1%	5.4%	5.5%	5.5%	5.7%
	Controlled guard invoc. (preferable)	0.0%	0.0%	0.5%	0.6%	0.7%	0.7%

Figure 9: Increases in execution time under the scenarios of controlled and uncontrolled guard invocations

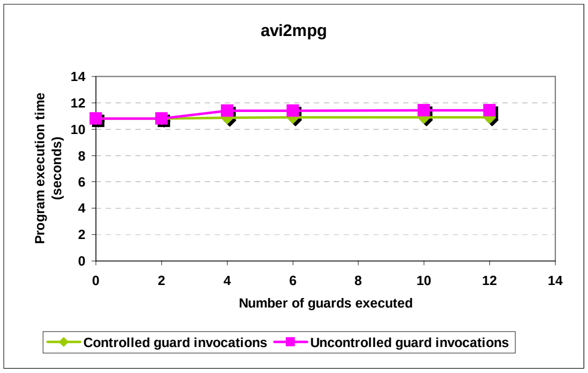


Figure 8: Comparison between the runtime performances of avi2mpg in two scenarios

gether. As a result, a software developer can have different copies of its software applications protected differently so that successful attacks against one of the copies would not work for the others (i.e., no "wholesale" attacks). We have developed techniques for preventing "diff" attacks that would compare two differently protected copies of the same software.

- **Configurable tamper-resistance**. The guarding approach makes it flexible for a software developer to control the levels of protection (e.g., how many guards) its software applications need, allowing configurable tamper-resistance with little performance degradation.

We have implemented a system that automates the process of installing guards in Win32 executables in a configurable manner. Our experiences have convinced us that it is possible to easily guard software which is difficult to "unguard"—i.e., asymmetry in the efforts (small effort to protect, large effort to attack).

Our results show that if configured appropriately, guards cause only slight impacts on the performance of guarded programs. We believe that such impacts are insignificant in most situations, and that they are reasonable tradeoffs for the levels of protection received.

We are currently in the process of completing Version 1.1 of our system. This version has the convenience of a graphical user interface integrated with Microsoft Visual C++ 6.0, and will extend the obfuscation capabilities of the current Version 1.0. Although the paper [3] gives theoretical evidence of the difficulty of absolute obfuscation, "practical" obfuscation (in the sense of delaying attacks on the software by substantially "raising the bar" for an attacker) are still a worthwhile endeavor in many practical situations. In our case what we really need out of obfuscation is limited to "code entanglement", that is, the binding of guard code with the original program's code so it is hard to disentangle them, that is, difficult to distinguish binary-level guard code from the original binary code (as mentioned in Section 2, there are many ways to achieve such binding, ranging from the use of artificially introduced dependencies and "dummy code", to the use of complex mathematical identities, etc). What we need is more limited, and experiments performed at Purdue and elsewhere lead us to believe that it is achievable in a practical sense. This implies that even if the regions of code containing guards were roughly located by an attacker, it would still be very difficult to "separate" and remove the guard code from the code needed by the program's functionality.

Additional work is also under way to port the system to other platforms, and to develop a facility that allows efficient and safe software patch distributions using the scheme described in this paper; here "efficient" is in the sense that the patch can have a small size compared to the total program, and "safe" in the sense that it is does not compromise the guarding network.

References

[1] David Aucsmith. Tamper-resistance software: an implementation. In Ross Anderson, editor, *Information Hiding – Proceedings of the First International Workshop*, volume 1174 of *LNCS*, pages 317–333, May/June 1996.

[2] T. Aura and D. Gollman. Software licence management with smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 75–85, May 1999.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO 2001*, August 2001.

[4] Clark Thomborson Christian Collberg. Watermarking, tamper-proofing, and obfuscation – tools for software protection.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, USA, May 1998.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand, 1998.

[7] Cloakware Corporation. Introduction to cloakware tamper-resistant software (trs) technology, March 2001. http://www.cloakware.com/pdfs/TRS_intro.pdf.

[8] Compuware Corporation. Numega softice. http://www.numega.com/drivercentral-/components/softice/si_features.shtml.

[9] H. G. Joepgen and S. Krauss. Software by means of the 'protprog' method. ii. *Elektronik*, 42(17):52–56, Aug. 1993.

[10] O. Kommerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proc. USENIX Workshop on Smartcard Technology*, Chicago, IL, May 1999.

[11] Josh MacDonald. On program security and obfuscation.

[12] Masahiro Mambo, Takanori Murayama, and Eiji Okamoto. A tentative approach to constructing tamper-resistant software. In *New Security Paradigms Workshop. Proceedings*, pages 23–33, New York, NY, USA, 1998. ACM.

[13] Landon Curt Noll, Jeremy Horn, Peter Seebach, and Leonid A. Broukhis. The International Obfuscated C Code Contest, 1998. http://www.ioccc.org/.

[14] A. Schulman. Examining the Windows AARD detection code. *Dr. Dobb's Journal*, 18(9):42,44–8,89, Sept. 1993.

[15] S. Smith and S. Weingart. Building a high-performache programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999.

[16] Simon Tatham and Julian Hall. Netwide Assembler. http://www.web-sites.co.uk/nasm.

[17] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, 12 2000.

[18] Steve R. White and Liam Comerford. ABYSS: An architecture for software protection. *IEEE Transactions on Software Engineering*, 16(6):619–629, June 1990.

[19] Bennett Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. pages 155–170, 1995.