Supporting A Dynamic Program Signature: An Intrusion Detection Framework for Microprcessors

Inoue, Koji
PRESTO, Japan Science and Technology Agency | Department of Informatics, Kyushu University

https://doi.org/10.15017/6356

出版情報：IEEE International Conference on Electronics, Circuits and Systems, pp.160-163, 2006-12. IEEE
バージョン：
権利関係：

Supporting A Dynamic Program Signature: An Intrusion Detection Framework for Microprocessors

Koji Inoue[†§]
[†]Department of Informatics, Kyushu University
6-1 Kasuga-Koen, Kasuga, Fukuoka 816-8580 Japan
§PRESTO, Japan Science and Technology Agency,
4-1-8 Honcho Kawaguchi, Saitama 332-0012 Japan
inoue@i.kyushu-u.ac.jp

***Abstract*—To address computer security issues, a hardware-based intrusion detection technique is proposed. This uses the dynamic program execution behavior for authentication. Based on secret key information, an execution behavior is determined. Next, a secure compiler constructs object code which generates the predetermined execution behavior at runtime. During program execution, a secure profiler monitors the execution behavior. If the profiler cannot detect the expected behavior, it sends an alarm signal to the microprocessor for terminating program execution. Since attack code cannot anticipate the execution behavior required, malicious attacks can be detected and prohibited at the start of program execution.**

I. INTRODUCTION

In modern computer design, system security is a very important consideration. Malicious programs such as computer viruses are particularly serious security threats. These attempt to invade computer systems and perform malicious operations without users realizing. Computers are widely used in various forms, such as desktop PCs and mobile devices like cellular phones. These computer systems deal with a lot of important information, such as electronic money. Therefore, attention must be paid not only to computer performance and power consumption but also to security.

A well known and popular approach to detect malicious programs is to search for attack code in the disk areas, i.e., virus scanning. In this approach, a virus definition database (a kind of black list) is used to find malicious programs. However, one of the main drawbacks of this static approach is that we cannot detect unknown attack code. Thus, it is impossible to find new malicious programs. However, a number of intrusion detection techniques have been proposed. The basic strategy is as follows: 1) extract characteristics of the authenticated application program from the source code, and 2) monitor the execution behavior at runtime. However, this method has limitations in detecting malicious attacks (as explained in Section II).

In this paper, a novel approach is proposed for detecting malicious attacks at runtime. Based on secret key information an execution behavior is determined, e.g., a memory access pattern. In this process, no characteristics of the target application program are considered, making the approach application independent. Next, object code is created which generates the determined execution behavior at runtime. During program execution, a secure profiler monitors the execution behavior. If the expected behavior is not seen, the profiler sends an alarm signal to the microprocessor for terminating execution of the current program. In effect, a dedicated execution behavior is regarded to be a program signature, and it is attempted to insert this signature into the authenticated program code.

This paper is organized as follows: Section II describes related work aimed at solving the problem of malicious programs. In Section III, an approach for generating dynamic program signatures is proposed, and strength of security is discussed. In Section IV, the performance/cost overhead of the approach is evaluated. Finally, Section V concludes this paper.

II. RELATED WORK

To attack vulnerable computer systems, at least two operations must be performed: the injection of attack code, and the hijacking of program execution control. A number of techniques to prevent malicious attacks have been proposed, and they can be categorized into two approaches: either they close security holes, or they execute only trusted (or authenticated) program code.

Some malicious attacks exploit security holes existing in computer systems. For instance, stack smashing is a well known technique for executing malicious code. Stack smashing is based on buffer overflow attacks, caused by writing an excessively large amount of data into a buffer. Unfortunately, the C programming language does not automatically perform array bound checks, thus a return address can be corrupted. This weakpoint exists mainly in the standard C library, in functions such as *strcpy()*. Therefore, many programs are vulnerable to buffer overflow. By exploiting the buffer overflow vulnerability, attackers attempt to change the return address value to refer to the start of a piece of injected malicious code. If these types of security holes are precisely known, countermeasures can be designed. For example, reference [8] formulates the detection of

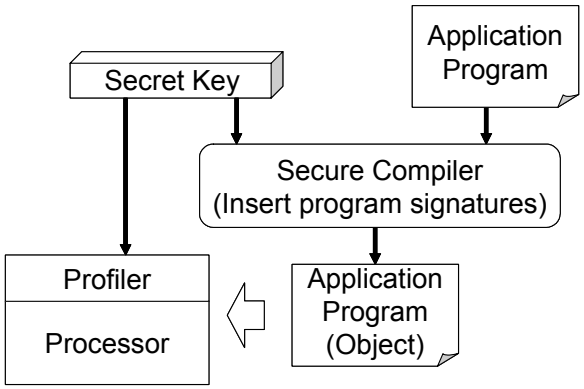

Figure1: Secure Computing Environment Framework

buffer overflows as an integer range analysis problem, in order to detect the possibility of stack smashing. *StackGuard,* which is a patch to gcc, is another static approach for defense against stack smashing [1]. Other techniques use hardware-based detection of return address corruption [4][5].

One of the drawbacks of approaches which close security holes is that they cannot deal with unknown security holes. The program recognition approach, on the other hand, does not have this problem. With this method, only authenticated programs are executed, based on a "white list". When an application program is executed, the computer system checks whether or not the program can be trusted. In order to achieve secure execution in this way, a number of intrusion detection system (IDS) techniques have been proposed [3][7]. These first extract the characteristics of the target (or trusted) program, for example the sequence of system calls. Then, the computer system monitors the execution of the program in order to confirm whether or not its behavior follows the extracted characteristics. However, an attacker can create malicious code which follows the characteristics of the trusted program. In this case, the malicious code will not be detected. Another straightforward way of authenticating program execution is to encrypt the program code with a secret key. Encrypting program code was originally introduced to help protect against software piracy, however it can also be exploited in execution authentication [2][6][9]. However, since all of the object code must be decrypted at runtime, this may cause significant performance degradation.

III. Inserting Program Signatures For Dynamic Execution Authentication

A. Main Concept

In a secure computing environment, we believe that dynamic program authentication should be supported in order to prohibit the execution of malicious code. As explained in Section II, the rules-based approach allows malicious code to run on the microprocessor if it satisfies the rules. This kind of attack code can be constructed if the attacker has the same source or object

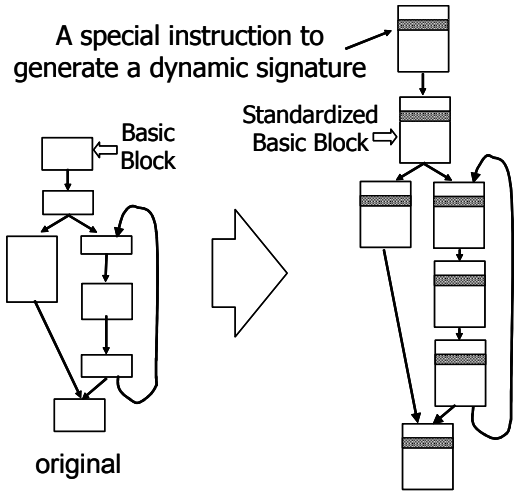

Figure2: Basic Block Standardization

code. On the other hand, although encrypting the trusted program code with a secret key may be efficient, the negative impact on processor performance is serious. In recent secure microprocessor designs, a large, safe on-chip cache is assumed. In this case, decryption of object code is required only when a block of data fills the cache. With this scenario, we can effectively nullify the negative impact on performance. However, in embedded processors which usually do not have large on-chip caches, the delay due to decryption greatly affects memory performance. Moreover, in multi-core processors, it is difficult to ensure that on-chip memories are secure, since they are shared by other processor cores. If a processor core is infected by a virus, this may attack the other cores.

To improve on conventional approaches, we propose an application independent methodology for runtime program authentication. Figure 1 shows the framework for our secure computing environment. In our approach, when the program source code is compiled, we insert a signature which consists of a sequence of execution events, e.g., memory access operations. This means that when the trusted program is executed, we see a precise predefined execution behavior. Of course, to ensure integrity of the approach, the compiler and the original source code need to be secure. A profiler monitors the events at runtime, and checks whether or not the execution follows the expected behavior. An implemented example is described in the next section

B. Implementing A Dynamic Program Signature Based On Memory Accesses

There are many aspects of execution behavior which can be considered candidates for generating a dynamic program signature. In this section, we discuss an example implementation which focuses on memory accesses. Here, we define a dedicated memory access pattern as a dynamic program signature. Namely,

the execution of the authenticated program causes a memory access at every *N*th instruction, the access address of which is the same as the secret key value. If the bit width of the secret key is larger than that of the memory address to be generated, we can use a hash function (i.e., a transformation) which takes an input and returns a string of fixed size. In this scenario, our secure computing environment works as follows:

- The compiler generates object code to create the dynamic program signature. Generally, execution behavior depends heavily on input data, thus it is usually difficult to anticipate execution behavior at compile time. To get around this, we make all the basic blocks the same size, as shown in Figure 2. Thus, all the basic blocks contain the same number of instructions. We insert one memory access instruction (load or store), the access address of which is set to the value of the secret key, into each basic block with the same offset. Therefore, regardless of branch results, a memory access for the dynamic signature always takes place at every Nth instruction. Note that the number N becomes "*the size of a basic block – 1*".
- During the execution of the authenticated program, the hardware profiler monitors memory access events. Also, it checks the number of instructions executed. In out-of-order high-end microprocessors, we need to monitor either the fetched instructions or the committed ones. If the profiler detects the correct dedicated memory access behavior, this means that the microprocessor is executing the authenticated program. Otherwise, execution control may have been hijacked by malicious code. In this case, the hardware profiler sends an alarm to the microprocessor to terminate execution of the current program.

C. Security Strength

In this section, we qualitatively evaluate the strength of security for the proposed method. In our approach, there are two monitoring criteria for authenticating the program execution: one is the memory access address of the key instructions, and the other is their execution interval. The former depends on the secret key and the latter is dictated via an option on code compilation. Thus there are at least two scenarios in which attackers can hijack the program execution.

If attackers can monitor the execution behavior of the authenticated program, they may create malicious code which has the same signature. This is because the memory access address generated by the key instructions and the interval of their execution can be determined easily. However, attackers cannot usually observe the execution behavior from outside the computer system. Furthermore, after each program execution, the original source code can be recompiled in order to insert a new signature, i.e., a different secret key and key instruction interval pairing. This strategy makes it extremely difficult to generate malicious code with the correct signature.

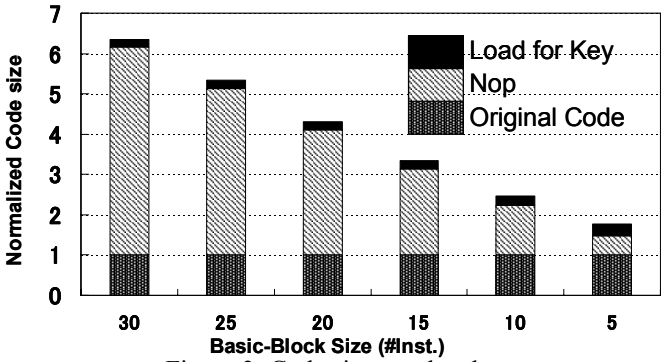

Figure 3: Code size overhead

The second way in which attackers can break through the intrusion detection mechanism is via interval monitoring. If the size of the attack code is less than the standardized basic block size, the execution of the attack code cannot be detected. However, a small size can be chosen for the basic blocks, for instance five instructions. This means that the attack code has to be less than five instructions, and satisfying this condition is extremely hard for attackers.

Another drawback of our framework is that currently it does not support signatures for dynamic libraries. In our framework, each program can have a unique signature in order to improve security strength. However, since a dynamic library is shared by many programs, we cannot ensure the consistency of the dynamic program signature. This issue can be resolved by using a special signature for the libraries. In this case, the profiler needs to distinguish between the execution of user code and the execution of dynamic libraries.

IV. Quantitative Evaluation

A. Experimental Set-Up

We developed a code translation tool which standardizes the basic block size and inserts a dynamic signature into the translated code. We also extended the SimpleScalar tool set [10] to support the profiler which monitors the memory access behavior. The SPEC95 benchmark program, 129.compress, was used for the simulation [11]. We assume an in-order embedded microprocessor, such as the StrongARM.

In this section, we measure the object code size and the program execution time with various basic block sizes. In order to statically control the execution behavior, as explained in Section III, our approach makes the basic blocks all the same size. If the basic block is initially too small, NOP instructions are inserted. If a small size is chosen for the basic block standardization, the total number of basic blocks in the full program code is increased. In this case, although the total number of NOP instructions needing to be inserted for basic block standardization tends to be reduced, the number of key

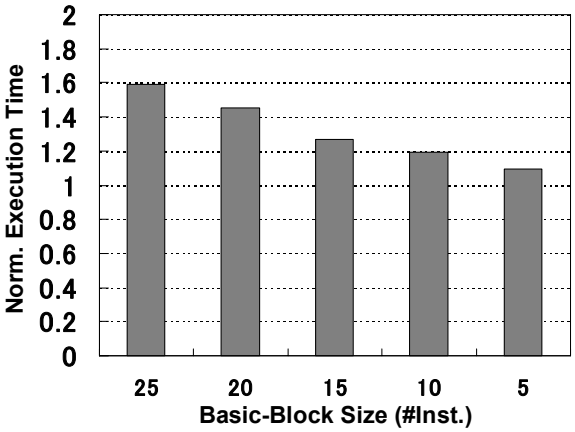

Figure 4: Performance overhead

instructions inserted is increased. This affects both the total code size and the program execution time.

B. Code Size Overhead

Figure 3 shows the code size and its breakdown when we vary the basic block size from 5 to 30 instructions. All results are normalized to the code size of the original object code. "Load for Key" refers to the memory access instruction which generates a dynamic signature. From the figure, we see that the code size is proportional to the standardized basic block size. This is because originally the average basic block size is around five instructions. This means that in almost all the cases we need to insert NOP instructions into each basic block. On the other hand, the increase in the number of key instructions is very small. The optimum basic block size is five instructions, and the code size increase is about 76.6%.

C. Performance Overhead

Figure 4 shows the program execution time. All the results are normalized to the performance of a conventional processor. From the figure, we see that reducing the basic block size alleviates the negative impact of basic block standardization. In the best case, where each basic block consists of five instructions, the performance overhead is only 9.6%. Almost all of the performance degradation comes from the execution of NOP instructions. However, we believe that this performance overhead is tolerable in order to realize secure computing.

V. CONCLUSIONS

In this paper, a dynamic program signature was proposed in order to prevent the execution of unauthorized code. In the framework described, the trusted program is translated into object code which generates a previously determined execution behavior at runtime. A hardware profiler monitors the behavior in order to check whether or not the microprocessor is currently executing an authenticated program. If the profiler detects unexpected behavior, it sends an alarm signal to the microprocessor to terminate execution of the program.

The cost and performance overheads incurred by introducing the dynamic program signature were evaluated. It was observed that the program code increases in size by 76.6%, and the performance overhead is 9.6%, compared with a non-secure conventional microprocessor. It is believed that a microprocessor must be able to recognize whether or not the current program execution has been authorized, and the ideas proposed in this paper make this possible. However, there are still some issues to be overcome in order to achieve secure computing. First, as explained in Section III, dynamic libraries must be catered for. Second, multiprogramming environments must be considered. For these, when a context switch takes place, the profiler needs to suspend monitoring of the dynamic program signature.

ACKNOWLEDGMENT

I would like to thank Prof. Shingi Tomita, Prof. Hiroto Yasuura, and all other members of the PREST "information infrastructure and applications" research group for discussions at technical meetings. This research was supported in part by Grant-in-Aid for Creative Basic Research, 14GS0218, and for Encouragement of Young Scientists (A), 17680005.

REFERENCES

[1] C.Cowan, C.Pu, D.Maier, H.Hinton, J.Walpole, P.Bakke, S.Beattie, A.Grier, P.Wagle, and Q.Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," Proc. of 7th USENIX Security Symposium, Jan, 1998.

[2] G. Edward Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas,"Efficient Memory Integrity Verification and Encryption for Secure Processors," Int. Symp. on Microarchitecture, pp.339-350, Dec. 2003.

[3] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff, "A Sense of Self for Unix Processes," Proc. of the 1996 IEEE Symposium on Security and Privacy (S&P), pp.120-128, 1996.

[4] K. Inoue, ``Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," ACM Computer Architecture News, vol33, no.1, pp.81-89, Mar. 2005.

[5] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," Int. Conf. on Security in Pervasive Computing, Mar. 2003.

[6] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[7] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," IEEE Symposium on Security and Privacy (S&P01), pp.156-168, May 2001.

[8] D.Wagner, J.S.Foster, E.A.Brewer, and A.Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," Proc. of the Network and Distributed System Security Symposium, Feb. 2000.

[9] J. Yang, Y. Zhang, and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," Proc. of the Int. Symp. on Microarchitecture, pp.351-360, Dec. 2003.

[10] SimpleScalar Tool Sets, http://www.simplescalar.com/.

[11] SPEC(Standard Performance Evaluation Corporation), http://www.specbench.org