A Heuristic Policy-based System Call Interposition in Dynamic Binary Translation

Deen Zheng, Zhengwei Qi, Alei Liang
School of Software
Shanghai Jiao Tong University
Shanghai, P.R.China
{nouligen, qizhwei, liangalei}sjtu.edu.cn

Hongbo Yang, *Haibing Guan
Department of Computer Science
Shanghai Jiao Tong University
Shanghai, P.R.China
{yanghongbo819, hbguan}sjtu.edu.cn

Liang Liu
IBM China Research Lab
Beijing, P.R.China
liuliang@cn.ibm.com

Abstract—**Dynamic Binary Translation (DBT) is a well known software technology that enables seamless cross-ISA execution. Unfortunately, many malicious programs that may lead to unauthorized access can run easily and unrestrictedly under the DBT system. Because these malicious programs must go through the system call interface to take malicious action, system call interposition has become a widely used technique for intrusion detection and prevention. In this paper, we present HPSCIBit, a solution that efficiently confines malicious applications, supports automatic policy generation and interactive policy generation, intrusion detection and prevention in the DBT system. The experimental result on SPEC2000 CINT benchmarks shows that HPSCIBit is an effective and low overhead solution to the cross-ISA security issues.**

Keywords-dynamic binary translation; system call interposition

I. INTRODUCTION

Over the years, dynamic binary translation (DBT) has become an increasing useful technique in virtualization area. It is a process of translating and optimizing binaries from one instruction set architecture (ISA) to another, for the purpose of running a program written for the one instruction set on the other one [1]. Several dynamic binary translators have been developed. Daisy [2] translates PowerPC instructions to VLIW instructions through the hardware support. UQDBT [3], a machine-adaptable dynamic binary translator, can support different source and target ISAs. Some systems like Dynamo [4] and DBO [5] use DBT to optimize the native binaries to improve performance, while DynamoRIO [6], Valgrind [7] and Pin [8] provide runtime instrumentation of the applications.

Despite many uses to which the DBT system has been put, it still remains a significant challenge to build secure DBT systems. Many DBT systems like QEMU [9] may introduce security vulnerabilities to the host system during the execution of malicious program because of much concern about the cross-ISA execution ability and the neglect of importance of secure running.

For the reason that security attacks must exploit the system call interface to take malicious action, system call interposition is a powerful method for intrusion detecting and preventing.

Strata [10] is a software dynamic translation system which can offer a basic safe virtual execution engine. It supports system call interposition by affording an API mechanism which specifies or adds system call policy through four API functions. One drawback of this way is that policies can only be added by the developer of Strata. Furthermore, writing policies through the API is complicated and inefficient.

This paper presents HPSCIBit, a solution that supports automatic and interactive policy generation, intrusion detection and prevention. HPSCIBit provides a portable, extensible system call interposition module which is based on the dynamic binary translation framework called CrossBit. It first captures an application's normal system call behavior based on having a policy and then halts execution if an application deviates from this normal behavior during execution. In this way, the malicious applications can be detected and the damage can be minimized. HPSCIBit is also designed to be easily ported to new platforms, and up to now it has been targeted to MIPS/Linux.

The remainder of the paper is organized as follows. Section 2 provides the details of implementation of HPSCIBit. In Section 3 we provide experimental results, including performance overhead and evaluation of the effectiveness of policy-based system call interposition. We then discuss related work in Section 4. In Section 5 we discuss future work and summarize this paper.

II. DESIGN AND IMPLEMENTATION

HPSCIBit is a framework implemented on CrossBit by adding an extensible system call interposition module. It is designed to detect and prevent intrusion from malicious application as far as possible by system call checking. We first

*Corresponding author.

This work was supported by The National Natural Science Foundation of China (Grant No.60773093,60873209), The Key Program for Basic Research of Shanghai (Grant No.08JC1411800) , The Ministry of Education and Intel joint research foundation (Grant No.MOE-INTEL-08-11), IBM SUR Funding and CRL JP Funding.

give an overview of HPSCIBit. Then, we describe the design of the system call interposition module in detail.

A. Overview of HPSCIBit

HPSCIBit is based on CrossBit [11, 12, 13] which is a resourceable and retargetable DBT system with an intermediate representation (IR). Until recently, it has fully supported the MIPS source platforms and the IA-32 target platform.

As shown in Figure 1, HPSCIBit consists of six components: bootstrapper, dispatcher, translation module, code cache (*TCache*), system call handler, and interposition module. First of all, the bootstrapper initializes the whole system environment and then loads the source executable image into the memory. After loading the image file successfully, the first basic block (*VBlock*) is built and translated to target binary form (*TBlock*) by the translation module. Then this *TBlock* is executed directly on target machine as well as being kept in the *TCache*. A mapping of the source and target address of the basic block (called *SPC* and *TPC*) is stored in a hash map. After execution of a *TBlock*, the *SPC* of the next basic block will be known, and this main work flow will be repeated until the termination of the program.

During the process, the dispatcher acts as the controller of the workflow. When the *SPC* of the next block is obtained, the dispatcher looks up the translated basic block in the *TCache* by the *SPC*. Actually, this look-up operation is to find out whether a *TBlock* exists in the *TCache*. If the *TBlock* is cached successfully, the look up operation returns the entry address (*TPC*) of TBlock for executing. And the next *SPC* will be returned to the dispatcher after the execution of this *TBlock*. And if the *TBlock* is not cached (called cache miss), this would lead a new translation process executed by translation module. When the translation of a *VBlock* is completed, the generated *TBlock* would be stored in *TCache*, and the Hash Map which stores *SPC* and *TPC* would also be updated.

In HPSCIBit, the system calls are handled by both the system call handler and the system call interposition module, as shown in Figure 1. Once a system call is encountered during the execution of a *TBlock*, the execution will be halted and the control will be transferred to the system call interposition module. This module is in charge of checking the system call, and we will give the details in the following section. If the behavior of the system call is malicious, the executing process will be terminated; otherwise, the control will then be transformed to the system call handler which is responsible for running the system calls on the target platform. Because the principle of passing parameters among various ISAs is different from each other, the system call handler has to transfer the system calls from the source platform to the target platform before the execution. Taking the system call read for example, Table 1 shows the transformation procedure from MIPS to IA-32.

B. System Call Interposition Module

The System call interposition module consists of two parts: policy generator and system call checker.

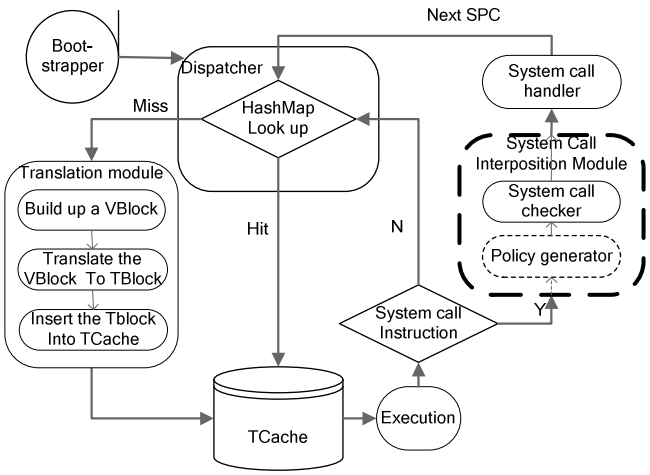

Figure 1. Framework of HPSCIBit

TABLE I. SYSTEM CALL TRANSFORMATION

System call	*MIPS*	*IA-32*
System call number	4003	3
First parameter(file descriptor fd)	a0	ebx
Second parameter(buffer)	a1	ecx
Third parameter(count bytes)	a2	edx
Return value	v0	eax

The policy generator produces a serial of policy for each system call both automatically and interactively. The system call checker ensures that each system call must match its policy.

DBT's ability to control the execution of a program through the runtime provides an ideal mechanism for implementing a system call interposition module.

To design a policy-based system call interposition, we first need an appropriate system call policy that specifies what must be satisfied when the system call is executed.

1) Policy Specification Design: A typical policy requires that a system call be constrained to a specified system call arguments.

We use a Systrace-like [14] policy language for our HPSCIBit. Each policy statement is a boolean expression *B* combined with an action clause: *B* then *action*. Valid actions include *permit*, *deny* and *ask*. If the boolean expression is true, the specified action is taken. The *ask* action requires the decision from the user to permit or deny the system call explicitly. Taking the MIPS ISA for example, a little part of the policy specification is listed in Figure 2.

2) Policy Generation Design: Once a detailed policy specification has been finalized, we generate the policy database that will be used for system call checking.

```
4003: filename eq "/etc/passwd" then deny
4003: filename eq "/etc/shadow" then deny
4003: filename eq "/Desktop/permit" then permit
4003: filename eq "/Desktop/ask" then ask
4005: filename eq "/etc/host.conf" then deny
4038: oldfilename eq "/etc/group" then deny
```

Figure 2. Part of the policy specification

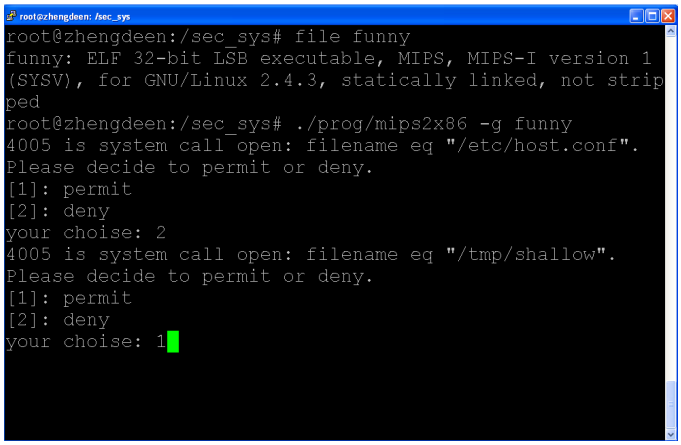

Figure 3. Interactive policy generation

In practice, two techniques can be used to generate policies: one is the manual way that the policies can be written by hand, the other is the training way that the policies can be learned by examining some sample runs of the program. Handwritten policies can be extremely precise, particularly when they are written by someone with deep knowledge of the program. However, they are difficult to produce and maintain. Unlike manual policy, training can be automatic, and also be easy to maintain. However, training by its nature does not examine all possible behaviors of the program. Here, we combine the two techniques to generate policies.

In HPSCIBit, we generate the policy by running a serial of executable binary files which don't contain malicious code. We translate the system call arguments of the source ISA, and canonically transform them into policy statements. Once an executable binary file attempts to execute a system call during the training run, it is checked against the existing policy and if not covered by it, a new policy statement that permits this system call is appended to the policy.

Using automatic policy generation, we can't enumerate all legitimate actions. However, it provides a basic policy that covers a subset of necessary actions. In conjunction with interactive policy generation, we can achieve more coverage of the policy.

Interactivity requires user's decisions when the current policy does not cover the attempted system call. As shown in Figure 3, with the "-g" command-line option, the user is notified by a consulting statement in our system call interposition module. User then can permit or deny the current system call invocation, and improves the current policy by appending a policy statement.

HPSCIBit provides global policies for all executable binary files, and these polices are stored in a specific directory.

3) System Call Checking Mechanism: Once the security policy of a source ISA has been finalized during the first execution, automatic system call checking would be employed for the next run. As shown in Figure 1, the policy generator is not called during the checking procedure. Unlike the policy generation, the user is not asked for a policy decision when an executable binary file attempts to execute a system call that is not covered by the policy during the runtime checking.

The checker first loads the policy file into the non-volatile memory. For each system call, system call checking starts at the beginning of the policy and terminates with the first boolean expression that is true.

The *action* after the true expression determines if the system call is permitted or denied. At runtime, each system call including the system call number and related arguments are verified against the policy in this way. If the system call matches the policy, it is allowed to be translated to the target system call; otherwise, the executing process will be terminated.

III. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance overhead induced by the system call interposition mechanism on the HPSCIBit with MIPS and IA-32 as the front-end and back-end respectively. We begin with a description of results from microbenchmarks that measure the overhead on individual system calls, and then measure the effect on the overall execution time for the SPEC INT2000 benchmarks with the reference input. The experiment is taken on a Linux platform with Intel Xeon CPU, 2.00GHz, and 4GB of memory for all the measurements. And all these benchmark programs were cross-compiled using mipsel-hdhlan-linux-gcc-3.3.1 into statically linked executable binary file.

A. Microbenchmark

Figure 4 shows the execution time of four microbenchmarks under HPSCIBit and CrossBit system. And these results were obtained by executing each system call 10,000,000 times using a loop, and measuring the real time. The read microbenchmark here is a loop of reading ten bytes from a file. And the write microbenchmark is also a loop of writing ten bytes to a file. Each microbenchmark was repeated 12 times, the highest and lowest readings were discarded, and the average of the remaining 10 readings is used.

The results indicate a noticeable cost for the system call checking mechanism, about 10%-90% overhead for each microbenchmark.

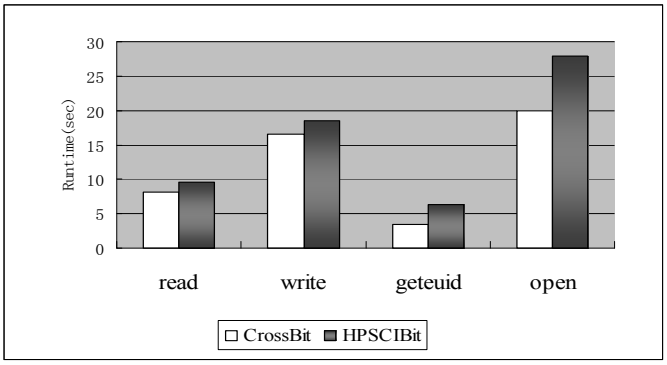

Figure 4. Runtime for microbenchmarks under CrossBit and HPSCIBit

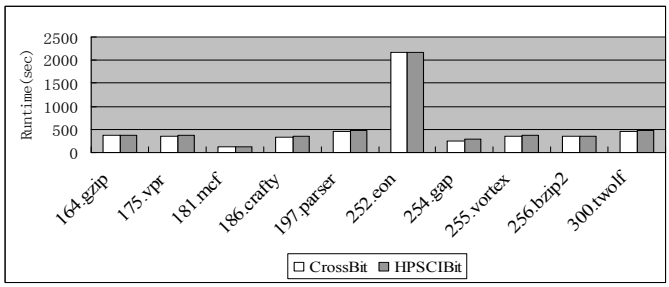

Figure 5. Runtime for SPEC INT2000 under CrossBit and HPSCIBit

These results are not out of expectation because of the large amount of loops in each benchmark compared to the normal programs.

B. *Large scale benchmark*

Figure 5 shows the execution time of SPEC INT2000 benchmarks HPSCIBit compared to the original CrossBit.

The results indicate a low overhead which is below 5% induced by the system call interposition. These results meet our goals. The user can use HPSCIBit for secure cross-ISA execution without pay a significant cost.

IV. Related Work

In this section, we discuss some related work both in the areas of dynamic binary translation and system call interposition.

A. *Dynamic binary translation*

Dynamic binary has become popular in recent years as a technique to deal with the problem of binary portability among different ISAs.

In recent years, dynamic binary translation has been used in various settings. Intel's IA-32 EL framework [15] provides a software layer to allow running 32-bit x86 programs on IA-64 machines without hardware support. Qemu [9] uses binary translation to emulate multiple source-destination ISA pairs. Transmeta Crusoe [16] uses on-chip hardware to translate the x86 CISC instructions to RISC instructions on the fly. Valgrind uses dynamic binary translation technique to supports dynamic binary analysis tools, such as shadow value tools and memory check tools. Strata [10] provides a software dynamic translation infrastructure to implement monitoring and safety checking such as system call interposition and stack smashing preventing.

B. *System call interposition*

Many researchers have proposed and implemented system call interposition layers. Janus, by Goldberg et al. [17], is one of the first system call interposition tools. It presents an architecture that restricts an application's interactions with the underlying operating system by interposition on the system calls made by the application via standard process tracing mechanisms. Systrace [14], is a system call interposition layers like Janus, implemented in Linux as a loadable kernel module. Mohan Rajagopalan [18], et al., presents an approach to implementing system call interposition based on authenticated system calls.

V. Conclusion and Future Work

This paper presented a policy-based system call interposition mechanism that supports automatic and interactive policy generation, intrusion detection and prevention in DBT systems. We argued that system call interception is a flexible and appropriate mechanism for intrusion prevention in a DBT system. This paper addressed important issues which were not covered by previous research in the area of binary translation. The translation of system call arguments into human-readable strings in the DBT system allows us to design the rules of policy specification. It also supports both automatic policy generation and interactive policy generation. We described the policy specification and policy generation that are simple and yet effective in catching a large number of known attacks with minimal impact on the performance of the DBT system.

We analyzed the performance of HPSCIBit and showed that additional affect on performance is acceptable and often observable by the user of a sandboxed application.

We believe that our HPSCIBit is a simple and comprehensive way to incorporate checks on the execution of binary file at the time of handling of system calls.

In future, we wish to apply this technique to other ISAs such as PowerPC and SPARC to our DBT system.

Acknowledgment

Many thanks to Ruhui Ma, Yuemei He, other BT members and the anonymous reviewers for their constructive comments and suggestions.

References

[1] M. Probst, "Dynamic Binary Translation," UKUUG Linux Developer's Conference 2002.

[2] E. Altman, M. Gschwind, and S. Sathaye, "BOA: The architecture of a binary translation processor," In Research Report RC21665 IBM T.J. Watson Research Center (2000).

[3] D. Ung and C. Cifuentes, “Dynamic binary translation using run-time feedbacks,” Science of Computer Programming Volume 60, Issue 2, Elsevier North-Holland, Elsevier North-Holland, 2006, pp. 189-204.

[4] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system,” In Proceedings of the ACM SIGPLAN Confe-rence on Programming Language Design and Imple-mentation (June 2000), pp. 1–12.

[5] E. Duesterwald, “Design and engineering of a dynamic binary optimizer,” Proceedings of the IEEE, Special Issue on Program Generation, Optimization and Platform.

[6] D. Bruening, T. Garnett, and S. Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimizations,” In Proc. International Symposium on Code Generation and Optimization (2003), pp. 265–275.

[7] N. Nethercote and J. Seward. “Valgrind: A program supervi-sion framework,” In Proc. of the Third Workshop on RuntimeVerification (RV’03), Boulder, Colorado, USA.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation,” In Programming Languages Design and Implementation 2005 (June 2005), pp. 190–200.

[9] F. Bellard, “QEMU, a fast and portable dynamic translator,” In Proc. of the SENIX Annual Technical Conference, pages 41–46, April 2005.

[10] K. Scott and J. Davidson, “Safe virtual execution using software dynamic translation,” In Annual Computer Security Applications Conference (2002).

[11] Y. Bao, A. Liang, and H. Guan, “Design and Implementation of CrossBit: Dynamic Binary Translation Infra-structure,” Computer Engi-neering. Vol. 33, No. 23, China, 2007, pp.100-101, 134.

[12] H. Shi, Y. Wang, H.Guan, and A. Liang, “An Intermediate Language Level Optimization Framework for Dynamic Binary Translation,” ACM SIG/PLAN Notice, vol-42(5), May 2007.

[13] The Valgrind Developers. www.crossbit.org

[14] N. Provos, “Improving Host Security with System Call Policies,” In Proc. 12th USENIX security Symp., pp. 257-272, 2003.

[15] L. Baraz, et al, “IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium-based systems,” Proceedings of the 36th International Symposium on Microarchitecture pp. 191-202 Dec. 2003.

[16] A. Klaiber, “The technology behind Crusoe processors,” Tech. rep., Transmeta Corp., January 2000.

[17] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, “A Secure Environment for Untrusted Helper Applications,” In Proceedings of the 6th Usenix Security Symposium, July 1996.

[18] M. Rajagopalan, M.A. Hiltunen, T. Jim, and R.D. Schilichting, “System Call Monitoring Using Authenticated System Calls,” IEEE Trans. Dependable and Secure Computing, vol. 3, no. 3, pp. 216-228, July-Sept. 2006.