Monitoring Library Function-based Intrusion Prevention System with Continuing Execution Mechanism

Yudai Kato, Yuji Makimoto[1], Hironori Shirai, Hiromi Shimizu[2],
Yusuke Furuya, Shoichi Saito, and Hiroshi Matsuo
Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya 466-8555 Japan
belem@mail.ssn.nitech.ac.jp

***Abstract*—Anomaly-based Intrusion Prevention Systems have been studied to prevent zero-day attacks. However these existing systems can't prevent mimicry attacks because of the inadequacy of monitoring accuracy. Moreover, they provide no continuity for monitored applications when they have been compromised. In this paper, we propose a novel Intrusion Prevention System named Belem that detects anomaly states by checking the ordering of library functions and has a Continuing Execution Mechanism to provide application continuity. We implemented Belem on Linux and evaluated it.**

***Keywords*-Intrusion prevention system, Monitoring library function, Continuing execution, Self-healing, Checkpoint**

I. Introduction

Zero-day attacks continue to increase. They attack vulnerabilities for which no solutions currently exist. Currently, *anomaly-based Intrusion Prevention Systems* (IPSs) protect systems from zero-day attacks. This kind of IPS uses a *Regular Execution Rule* (RER) to classify system activities. Anomaly-based IPSs detect malicious attempts by running applications to confirm whether executing functions conform to the RER. Therefore such an IPS can detect malicious attempts by a conflict between the rules and the functions.

Moreover, there is a problem to continue services until the vulnerabilities are fixed. Therefore we provide not only IPS but also *a Continuing Execution Mechanism* (CEM) to detect and monitor vulnerabilities until they have been fixed. For these reasons, we propose Belem, which is a new high detectable IPS with CEM.

Many attacks change the process behavior defined in the executable files of a process. For this reason, an anomaly-based IPS makes a RER from source code, executable files, and a regular execution and monitors the executions using the rule. Many systems [1], [2], [3] use system-call hooking for process monitoring. This technique is generally considered effective because a system-call is essential for attacks that are intrusions into other systems or are tampering with important files.

However these systems cannot check a function that doesn't make any system-calls. They can only check limited functions that are called at that time; they cannot check exited functions. For this reason, they can't detect *mimicry attacks* [1] that ape regular execution.

To combat such attacks, we propose **Belem**: (*Belem is Effective Library Executing Monitor*. Belem has two mechanisms, IPS and CEM, and is composed of a user library (libelem), a modified kernel for Belem, and a RER. The RER is obtained by analyzing an executable file of a process and makes normal execution orders of library functions. Belem detects illegal library calls by the RER. This check allows Belem to monitor a process more precisely. Moreover Belem continues the execution of a monitored process when its IPS detects an intrusion. Generally, a server system cannot provide service until the server vulnerabilities have been corrected. Belem's CEM avoids the monitored process from stopping or rebooting by the IPS when the process is attacked. CEM is a kind of a self-healing mechanism that improves application availability.

This paper is organized as follows. Section II describes related researches and how the IPS monitors a process. Section III describes an overview of our approach. Section IV describes Belem implementation, and Section V evaluates Belem. We discuss our approach in Section VI and provide a conclusion in Section VII.

II. Related work

In this section, we describe anomaly detection methods and related works of CEM. An anomaly detection method monitors the execution of a process and compares it with a regular execution rule. Process monitoring methods can be classified into *system-call monitoring* (SCM) and *library function call monitoring* (LFCM).

A. System-Call Monitoring (SCM)

SCM compares called system-calls with a RER. Below, we discuss the problems of this method.

1) Overview of existing SCM systems: Wagner's [1] approach determines whether the order of the system-calls is in accordance with the rule whenever they are executed. However, this approach doesn't use any information except the order of system-calls; it becomes difficult to uniquely

[1]Presently with Mitsubishi Electric Corporation
[2]Presently with Toshiba Tec Corporation

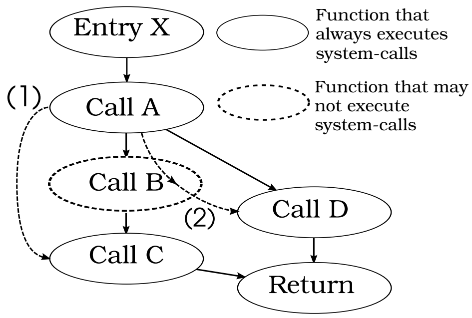


Figure 1. Problem of execution rule on related researches

identify a process execution state. Therefore an attacker can easily impersonate a regular execution. Feng [2] and Abe [3] confirm the return address in the call stack and accurately determine the execution state of a process when it executes a system-call besides Wagner's method . Moreover, checking the changes of the call stack from when the previous system-call was executed can monitor execution states that cannot be determined by only the order of the system-calls. However, functions that can be monitored by their systems are limited to called functions when a system-call is executed. Therefore, their systems cannot determine functions that didn't make any system-calls.

2) Problem of SCM: We explain the above SCM problem here. A system's RER defines the order of functions. Consider the case where the RER for Function X is expressed as in Figure 1, which shows that Function X calls functions in the order of A→B→C or A→D. X exits after that.

Function A was called from Function X and executed the n_{th} system-call, and Function B is called from Function X and executes the $(n+1)_{th}$ system-call, and we assume that Function B is called after Function A. Therefore, this call order corresponds with the RER of Function X. Now consider a situation where Function B doesn't always execute system-calls and the calling of Function B can't always be confirmed. In this situation, if the SCM system observes Function C at a $(n+1)_{th}$ system-call, it cannot verify the calling of Function B. Therefore, a RER of the system must include an extra rule so that it doesn't treat this situation as an anomaly. The extra rule accepts function calls in the order of A→C (Figure 1(1)) as well as A→B→C, as if they were present from the beginning. The rule of SCM systems can represent the detailed states of a process. However SCM systems cannot detect detailed states in runtime as the rule. Here, consider an attack that modifies the return address of Function B while executing Function B and calls Function D after Function B. This attack can be detected if Function B executes one or more system-calls because the RER doesn't define the order of the functions as A→B→D. However if Function B doesn't make any system-calls, the observed order is A→D, which matches the rule. As a result, the SCM systems defend themselves from this attack.

B. Library Function Call Monitoring (LFCM)

LFCM verifies that the calling of a library function is consistent with a RER. Compared to a SCM system, a LFCM system can monitor a detailed state and strictly define a rule. Therefore, a LFCM system can detect such illegal transfers as shown in Figure 1(2). However, no anomaly-based IPSs are perfectly based on library function calls. E-NeXSh [4] only uses LFCM for verifying system-calls. Moreover, e-NeXSh executes a special system-call for verifying each library function call, and its overhead costs are high. For reducing overhead, e-NeXSh limits the verified library functions to those most likely to be exploited by attackers. For this, it monitors as accurately as does system-call monitoring. Therefore, it may not be able to detect mimicry attacks.

C. Related work of CEM

Next we describe the existing continuing execution systems. Reactive immune systems [5] store all the modified data of the entire memory and the files and restore them when the system detects an anomaly. This system lightens its load by conjecturing vulnerabilities. However, the conjecture method is simply that a fixed-length array is considered vulnerable. This system has two problems. The first is that once the conjecture comes off, every time conjecture at the same point will come off. The second is that vulnerability cannot be conjectured, except for a fixed-length array.

Rx [6] periodically creates a process checkpoint and returns the process to the checkpoint at which it detected the anomaly. Rx modifies the environment and re-executes a doubtful function in the process to inspect the vulnerabilities. Rx, Sweeper [7], and ASSURE [8] periodically create a checkpoint and automatically seek vulnerability by re-execution from the latest checkpoint at which an intrusion was detected. However, these systems cannot find vulnerabilities that are placed before the latest checkpoint. In these systems [6], [7], [8], detecting anomalies, which is the starting point of restoring action, is assumed to be a relatively simple method, such as Exception Interrupts , CCure [9], StackGuard [10], and so on. Therefore some anomalies cannot be detected.

III. Proposed method

In this section, we propose Belem's IPS method that is based LFCM and CEM for a process when it is attacked. We start with the proposed IPS and next discuss the proposed CEM.

A. Monitored Library Functions

Many monitored library functions can accurately specify a process state. However, since an increase of monitored library functions will greatly increase the monitoring overhead, the number of monitored library functions must be set adequately. E-NeXSh only monitors functions that attackers use frequently. However it is not always ensured for every application, and some functions are inadequate, as already mentioned in II-B. Therefore, Belem targets all system-calls to monitor any execution states of a process and monitors all library calls that may execute system-calls. This approach enables Belem to capture a library function call even if it doesn't execute any system-calls. Belem can monitor more execution states than the SCM systems and e-NeXSh.

B. Storing an Execution History in User Space

As mentioned above, increasing the number of monitored functions increases overhead. E-NeXSh also uses a special system-call to record library function calls. Therefore, a special system-call is executed whenever library functions are called, and the overhead is larger than when only a library function is called. This overhead problem can also be applied to Belem, Because Belem needs to records a *call stack history* (CSH) as the library function was called for verification of the execution path at the subsequent system-call. To cope with the overhead, Belem stores CSH in the user memory space instead of kernel space. This way of storing history avoids Belem from calling any kind of special system-calls. Therefore, Belem can monitor the calling of all library functions that execute system-calls with small overhead.

C. Continuing Execution Method (CEM)

When an attack is detected, the source of the vulnerability is supposedly located before the attacked point. These systems [6], [7], [8] need to re-execute from the latest checkpoint to detect the vulnerable point. However, Belem has RER recoding all of a program's control flow. For this RER, Belem can precisely identify the attacked point and the previously executed process flow. Exploiting this rule, Belem estimates a candidate vulnerable point without re-execution. However, since it doesn't have an analysis function and cannot determine the vulnerable point, it expects and decides a scope containing vulnerability (vulnerable scope) and protects the process by Checkpoint-Rollback. A vulnerable scope starts from functions (read, recv, recvfrom, getenv) concerned with input just before an anomaly was detected and ends where it was detected. Initially, this vulnerable scope is large, but it is optimized during each repeated attacks, so the scope eventually becomes very small. Belem can effectively continue the process execution without excessive overhead. In CEM, a return address and malicious input data are checked, and the process state is restored by a rollback when abnormalities are detected. CEM prevents process behavior from being changed by overflow and code injection attacks. However, the CEM overhead described above is expensive. But it is only performed in a vulnerable scope.

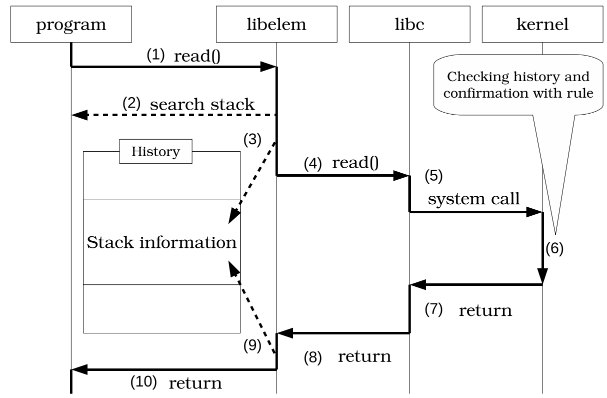


Figure 2. Process flow of IPS in Belem

IV. IMPLEMENTATION

Here, we describe a Belem implementation for Linux. Belem is composed of a *Belem kernel*, which is a modified Linux kernel, and *libelem*, which is a library hooking library function. The Belem kernel modifies an entry of system-call (entry.S) to hook and verify all system-calls and call stacks with a RER. The libelem hook library function calls with linker option LD_PRELOAD and makes a call stack history. In this section, we summarize a RER that provides normal execution order. Next, we describe the LFCM details by Belem with the generation of a CSH and the confirmation of a RER. Finally, we describe CEM.

A. RER Summary and Generation

A RER is generated from analyzing a disassembled executable file of a target program. It stores an execution order of the user functions and library functions for each user function. An entry of each user function in the RER contains all callee functions that the user function calls and the caller addresses that summon the callee functions.

B. Call Stack History and Confirmation of Execution

Figure 2 represents an overview of Belem and shows its process flow when the read library function is called. Belem adds a new CSH and confirms the RER and the CSH. We discuss these processes in this part.

1) Generation of call stack history: Libelem generates a CSH after hooking a library function (Figure 2(3)). Figure 3 shows an example of CSH. Figure 3(a) shows a RER and (b) shows three examples of CSH after executing (a). RER (a) prescribes that the main function is composed of three Library Functions, A, B, and D, and one User Function, C. Library Function E is called from User Function C.

Library Functions A and D always execute system-calls, but not Library Functions B and E. Figure 3(b) shows three executed results of Figure 3(a): (1) A→B→C→E→D, (2) A→C→E→D, and (3) A→D. (1) and (2) are normal CSHs but (3) is abnormal.

An executing history is added to a CSH whenever a library function is called. The history is removed from the CSH after the Belem kernel confirms that the RER and the history are the same when the next system-call is executed. Therefore a *confirmation and deletion* interval exists between successive system-calls, as shown in the dotted lines in Figure 3(b). The contents of the executing histories in a CSH are composed of all addresses calling callee functions in all functions between the main and current functions. These addresses are extracted from a return address from the call stack when a corresponding library function is called; these addresses are simply denoted by the function name in Figure 3. For example, Library Function A is directly called from the Main Function, and the CSH contains only one return address of Library Function A. Next, Library Function E is called from User Function C; therefore the history consists of two return addresses of User Function C and Library Function E. In this way, a CSH is recorded for all library function calls. At the end of calling a library function (Figure 2(9)), a LD_PRELOAD routine records the end of a library function to the history as well.

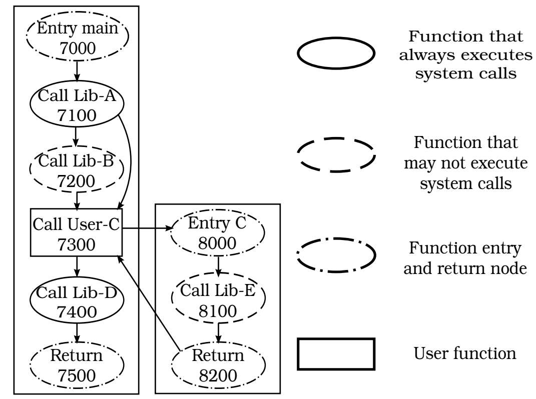
2) Confirmation of RER: The Belem kernel confirms that the CSH matches the RER after hooking a system-call (Figure 2(6)). If the confirmation fails, Belem detects the irregularity. For example, Figure 3(b)(3) is an irregular history, where Library Function D is called after Library Function A, but the RER does not include Function D in the next functions of Library Function A. Therefore Belem can detect that callee Library Function D is irregular. This detection ability is a feature of Belem's LFCM.

C. Continuing execution method

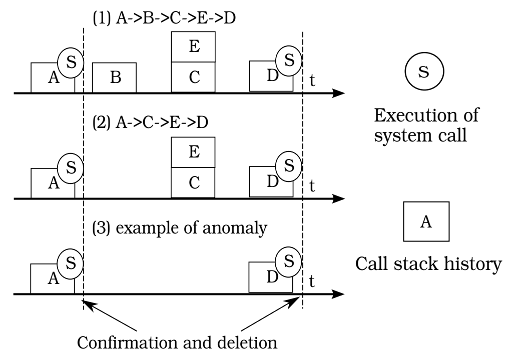
The CEM flows are shown in Figure 4. Figure 4(a) shows the CEM flow before the IPS detects an intrusion. If no intrusions are detected, CEM will do nothing. When IPS detects an intrusion, it sets a vulnerable scope and activates CEM. The end of the vulnerable scope is placed at detecting point, and the start of the scope is placed at the input function just before the detecting point.

Figure 4(b) shows the CEM flow after IPS detects an intrusion. For creating a checkpoint, Belem executes an original system-call that resembles a fork system-call. After detecting an intrusion, the flow mainly consists of two parts: process state confirmation in a vulnerable scope (*Detection Block* in Figure 4(b)) and restart processing when Belem previously failed to setup the vulnerable scope (*Restart Block* in Figure 4(b)).

In the Detection Block, the following three analyses are performed for each library function before execution:



(a) Example of RER



(b) Examples of CSH

Figure 3. Change of CSH and confirmation of RER

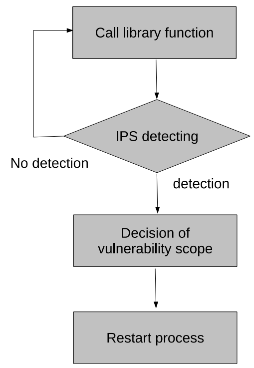
- Detect a tampered return address
- Detect an overflow of input data
- Confirm that input data don't contain any shell codes

Execution of the library function is continued if an anomaly is not found by the analysis. If an anomaly is found, the function being executed now is specified for the vulnerable point. In addition, Belem discards the input data and roll-backs to the adjacent checkpoint to resume execution.

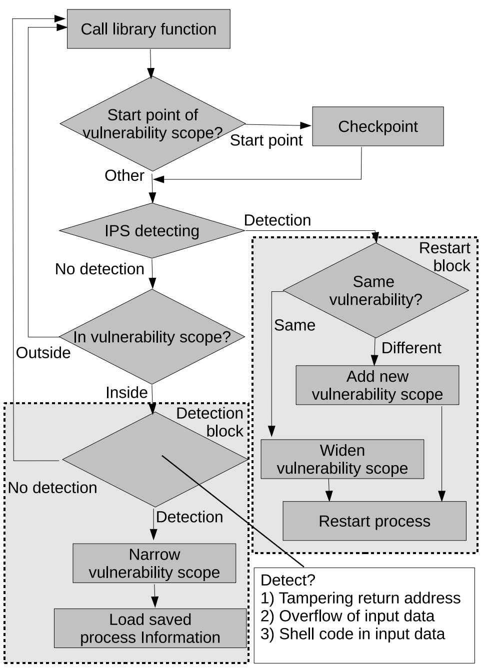
A restart block is executed when IPS detects a new attack. IPS judges that the current vulnerable scope is inappropriate and expands the scope if the detected point is the same as the previous detected point. On the other hand, if the detected point is not the same as the previously detected point, Belem judges that there is another vulnerable point and adds a new vulnerable scope. In this case, the process must be re-executed.

V. Evaluation

Next we evaluated Belem by two methods. The first is detection accuracy by an average number of candidate entries to which they can be transferred from the current entry in the rule. The second is monitoring overhead, which



(a) Flow of CEM before detecting intrusion



(b) Flow of CEM after detecting intrusion

Figure 4. CEM flow

is measured in both cases with and without CEM. The applications for evaluations are `wc` (a file utility) and `inetd` (a network server).

A. Evaluation of Detection Accuracy by Average Number of Transfers

We evaluated detection accuracy by the *Average Number of Transfers* (ANTs). An ANT is obtained by dividing the

Table I
AVERAGE NUMBER OF TRANSFERS OF PARTLY KEEPING TRACK METHOD (E-NEXSH)

Application	Average transfers	Total transfers	Total nodes
wc	1.57	367	234
inetd	2.01	783	389

Table II
AVERAGE NUMBER OF TRANSFERS OF FULLY KEEPING TRACK METHOD (BELEM)

Application	Average transfers	Total transfers	Total nodes
wc	1.33	312	234
inetd	1.48	574	389

Table III
MONITORING OVERHEAD

Application	Normal execution	Belem
wc	1.69 msec (1.00)	3.33 msec (1.97)
inetd	2.53 msec (1.00)	3.39 msec (1.34)

In parentheses, proportions of increase for normal execution

number of transfers in the rule by the number of nodes in it. A small ANT means that the transferable candidate nodes are rigidly restricted, and the execution flow of the process is also restricted. Therefore it is difficult to attack without being detected in small ANTs. We evaluate how accurately Belem can detect an anomaly by tracking the function calls and comparing them with an ANT in cases where library function calls cannot be always detected.

Here, we label the method, which can completely track library function calls, the *fully keeping track method* (FKTM), and the method that cannot completely keep track of library function calls the *partly keeping track method* (PKTM). The number of transfers increases in the PKTM, as shown in Figure 1. Belem adopted FKTM. PKTM was adopted by the SCM systems and e-NeXSh whose detection is as accurate as the SCM system.

The ANTs of PKTM and FKTM are shown in Table I and II, which indicate that both the average and the total number of FKTM transfers are less than that of PKTM. Therefore FKTM restricts a process flow more rigidly than PKTM. Such FKTMs as Belem can restrict the flexibility of an attack, increasing the safety of monitored processes.

B. Measuring Monitoring Overhead

One characteristic of `wc`, which is a file utility, is that it rarely executes system-calls, but it does call the library functions of each character in a file. When `wc` reads a 15 MB file, there are only 1000 system-calls, even though there are 15 million library function calls . On the other hand, the numbers of calling library functions and system-calls are almost the same in `inetd`: about 1000 times for each. We evaluated the effect of hooking a library function in the monitoring library function method using these applications.

Table IV
OVERHEAD OF CEM

Without monitoring	Monitoring
6 usec	416 usec

The evaluation of monitoring overhead is done without CEM, so we accurately measured the effect of hooking.

We measured the overheads of monitoring `wc` and `inetd` on the machine with Intel Pentium4 2.4GHz processor and 512MByte of RAM running Fedora Core 5 with kernel 2.6.17.8, and the result is shown in Table III. The result of `wc` is measured when it reads a 15 MB file. The result of `inetd` is measured when another host connects to it 1000 times. `Wc` shows that monitoring overhead increases significantly, if there are many more library function calls than system-calls. However, if the number of both calls is about the same, `inetd` shows that monitoring overhead is about 30%, which is considered acceptable for today's high-performance computers.

C. Overhead for CEM

We measured the CEM overhead and evaluated it by manually setting a vulnerable scope that only contained one library function. In this evaluation, Belem rollbacks the monitored process to the latest checkpoint whenever the library function is called. We measured the overheads of IPS and CEM on the machine with Intel Pentium4 3.0GHz processor and 1GByte of RAM running CentOS 5.4 with kernel 2.6.17.8, and show the monitoring overhead result for each vulnerability scope in Table IV. When Belem didn't monitor the process, the execution time was about 6 usec, and when it did monitor the process, the execution time was about 400 usec. From this result, the monitoring overhead for each vulnerability scope is estimated to be about 400 usec. The monitoring overhead details are that the overhead at the IPS is about 200 usec and about 130 usec for the fork operation as a checkpoint. CEM is only performed in a vulnerable scope after the IPS detects an attack, and the scope is narrowed after multiple attacks. For this mechanism, the Belem performance should be adequate in acute situations where threats such as zero-day attacks exist. Future works include decreasing overhead by improving the checkpoint method.

VI. CONSIDERATION

In this section, we discuss how to defend attacks directed at Belem by imagining a situation where an attacker invades a process after learning the process is protected by Belem.

A. Mimicry Attack

Attacks that mimic regular execution tamper with the call stack and/or the call stack histories and cause Belem to misjudge that the victim process runs along the regular execution rule. We discuss these mimicry attacks below.

Attacks disguised as call stacks exist [11]. Therefore, an attack disguised as a CSH against Belem is possible. However, realizing such attack requires that the call order of library functions, return addresses, and caller addresses must conform to those of the RER. Moreover, system-calls that attackers can execute are limited to what is given in the rule. Therefore, the actions that attackers can mount are restricted.

Belem makes a CSH each time a library function is called. If an attacker tampers with the call stack that cannot be transformed from the previously confirmed state, this malicious attempt can be detected by the inconsistency between the CSH and the RER. For this reason, tampering with the call stack before the previous library function call is impossible. Next, Belem confirms and deletes the CSH each time a system-call is executed. This produces a tampered CSH that is limited to one CSH after the previous system-call. If another history is tampered with, an attack can be detected by the inconsistencies between the RER and the CSH. Consequently, for any situations, the execution state cannot be returned to its state before the previous system-call. Tampering attacks are limited within the interval of consecutive system-calls.

Against tampering with the call stack, we can decrease the possibility of such tampering by concealing the CSH's address by randomizing it. However, since libelem needs the address to update the CSH, it must have the address itself. But the address must also be concealed. To defend against this attack, we randomize the libelem address and use the GOT [12][13] method. We prohibit references to the memory layout of the process by a proc file system to prevent leaks of addresses to other processes. CSH can be protected by these solutions.

As discussed above, some attacks may occur by tampering, but the operations that can be performed by attackers are limited. Moreover, Belem restricts the possible transferring within the interval of consecutive system-calls and library calls. This greatly complicates attacks. In addition, the randomizing memory layout parries attacks by making it difficult to determine the return addresses.

B. Invalidating attacks against Belem's protection mechanism

Belem needs correct RERs to detect intrusions because some attacks may invalidate the IPS by making the rule incorrect.

There may be an attack against Belem that modifies the RER to perform arbitrary attacks. If the modifications succeed, a malicious code can call any library functions needed for its attempts. To prevent this attack, an executable binary with a RER is combined and signed. The combined file is signed by those who compiled the execution binary or who can confirm the identity of the binary (e.g., a distributor of a Linux package). Public keys, which are needed for electronic signatures, can be acquired from a web site or a directory from those who compile the code. Belem can protect a distributed RER from this.

To prevent tampering with the RER while the corresponding binary is running, the Belem kernel prohibits anyone from writing to an address range where the RER is being loaded. Moreover, against changing the access right by mprotect(), Belem kernel prohibits changing the access right of the address range. These countermeasures allow us to protect the RER when the corresponding binary is running.

VII. Conclusion

We discussed Belem, which has high detectable IPSs and CEMs. It can inspect process states in detail by hooking library functions and decrease monitoring overhead by storing a CSH in a user memory space. Since Belem can track more detailed order of function calls than the existing systems, it can defend itself from an attack that tampers with executing states as regular states and protect a computer from threats such as zero-day attacks.

In the evaluations, the increase of the execution time when Belem monitors execution was only about 30%, which is acceptable overhead. We evaluated the ANTs of applications to show that Belem can restrict the transferring more rigidly with FKTM.

We described our CEM with a RER. CEM can grasp an accurate flow of execution before an attack without re-execution.

Our future work will examine our system with practical exploits and support transferring caused by a function pointer and longjump.

References

[1] D. Wagner and D. Dean, "Interusion detection via static analysis," in *IEEE Symposium on Security and Privacy*, 2001, pp. 144–155.

[2] H.H.Feng, O.M.Kolesnikov, P.Fogla, W.Lee, and W.Gong, "Anomaly detection using call stack information," in *IEEE Symposium on Security and Privacy*, 2003, pp. 62–77.

[3] H. Abe, Y. Oyama, M. Oka, and K. Kato, "Optimization of intrusion detection system based on static analyses (in japanese)," *IPSJ Journal*, vol. 45, no. SIG 3(ACS 5), pp. 11–20, 2004.

[4] G. S.Kc and A. D.Keromytis, "e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005, pp. 288–302.

[5] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 149–161.

[6] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 39, no. 5, pp. 235–248, 2005.

[7] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song, "Sweeper: A lightweight end-to-end system for defending against fast worms," in *In EuroSys' 07*, 3 2007.

[8] S. Stelios, L. Oren, P. Carlos, V. Nicolas, N. Jason, and K. A. D., "Assure: automatic software self-healing using rescue points," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009, pp. 37–48.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 232–244.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Conference*, jan 1998, pp. 63–78.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *14th USENIX Security Symposium*, 2005, pp. 161–176.

[12] U. Drepper, "How to write shared libraries," http://people.redhat.com/drepper/ dsohowto.pdf, 2006.

[13] Y. Furuya, S. Saito, and H. Matsuo, "Implementation of protecting program behavior rules for intrusion prevention system (in japanese)," in *IPSJ SIG Technical Report*, vol. 2009-OS-112, no. 7, 2009.