九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Secure Cache: Run-Time Detection and Prevention of Buffer Overflow Attacks

Inoue, Koji
Department of Informatics, Kyushu University

http://hdl.handle.net/2324/6232

出版情報：SLRC 論文データベース, 2005-07
バージョン：
権利関係：

KYUSHU UNIVERSITY

Secure Cache: Run-Time Detection and Prevention of Buffer Overflow Attacks

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
Tel : +81-92-583-7642
Fax : +81-92-583-1338
E-mail : inoue@i.kyushu-u.ac.jp

Abstract—This paper shows a novel cache architecture, called SCache, to detect and prevent buffer-overflow attacks at run time. A number of malicious codes exploit buffer-overflow vulnerability to alter a return-address value and hijack the program-execution control. In order to solve the security issue, SCache generates replica cache lines on each return-address store, and compares the original value loaded from the memory stack with the replica one on the corresponding return-address load. The number and the placement policy of the replica line strongly affect both performance and vulnerability. In our evaluation, it has been observed that SCache can protect more than 99.3% of return-address loads from buffer-overflow attacks, while it causes negligible performance overhead.

I. INTRODUCTION

Defending own computer system is one of the most important issues that should absolutely be solved for constructing pervasive or ubiquitous computing environment. Although the internet is a much useful instrument, it also gives an opportunity for attacking remote connected devices to malicious persons. In reality, a number of attackers attempt to distribute malicious codes via e-mail, web, file transfer service, etc. Therefore, we must pay attention to protecting own computer system.

A number of programs already integrated into existing computer systems potentially have *buffer-overflow vulnerabilities*. The overflow makes it possible to achieve two processes at the same time: injecting a malicious code and hijacking the program-execution control. This is one of the main reasons why the overflow vulnerabilities are commonly exploited. For example, the Code Red warm in 2001 and the Braster in 2003 that raged in the world utilize this defect. In 2001, 50% of CERT advisories relate to this weak point. Consequently, it is clear that detecting and avoiding the buffer-overflow attacks is very important to develop a secure computer system.

In this paper, we preesnt a novel cache architecture, called *Secure Cache (SCache)*, against buffer-overflow attacks. The attackers attempt to alter the procedure return address by causing a buffer overflow which breaks the structure of memory stack. SCache detects such return-address corruption without any software supports at run time. When a return address is pushed onto the memory stack, SCache generates one or more replicas of the return-address value, and saves them into the other cache space. We also show implementation alternatives which differ in the number and placement algorithm of the replica data. The security efficiency of SCache is evaluated by performing cycle-accurate out-of-order processor simulations. As a result, it is observed that we can protect more than 99.3% of return-address loads from the threat of buffer-overflow attack. The detail of this scheme was discussed in [7].

This paper is organized as follows: Section 2 explains the mechanism of buffer overflow and shows related work to solve this issue. Section 3 proposes the SCache architecture. In Section 4, we evaluate the vulnerability of several SCache models. Performance and energy overhead are also evaluated in this section. Finally, in Section 5, we conclude this paper.

II. STACK SMASHING

A. Buffer-Overflow Attacks

To attack vulnerable computer systems, at least two processes have to be done: injecting an attack code and hijacking the program-execution control. The buffer overflow makes it possible to achieve both the issues simultaneously. The buffer overflow is caused by writing an inordinately large amount of data into a buffer. Unfortunately, the C programming language does not perform automatically array-bound checks, and this defect mainly exists in the standard C library such as *strcpy()*. Therefore, many programs have the possibility to suffer from the buffer-overflow vulnerability.

The overflow breaks memory stack structure as depicted in Fig. 1, so called *stack smashing*. In this figure, we assume that the function *f()* calls the function *g(),* which includes a vulnerable operation *strcpy()*, as shown in the rightmost figure. The state of the memory stack immediately after the function call *g()* is depicted in the leftmost figure. The stack consists of the function parameters, the return address to the caller, the previous frame pointer, and the local array variable *buf*. In *g()*, if the size of the string pointed by *s* is larger than the memory size allocated for the local variable *buf*, a buffer overflow takes place when the *strcpy()* function is executed. As a result, the contiguous stack contents are overwritten. If the pointer *s* points a malicious string which is meticulously constructed by an attacker, the attack code is injected into the memory stack and the return address is altered to the top of the injected code, as shown in the middle one in Fig. 1. The corrupted return address is set to the program counter (PC) when the execution of *g()*

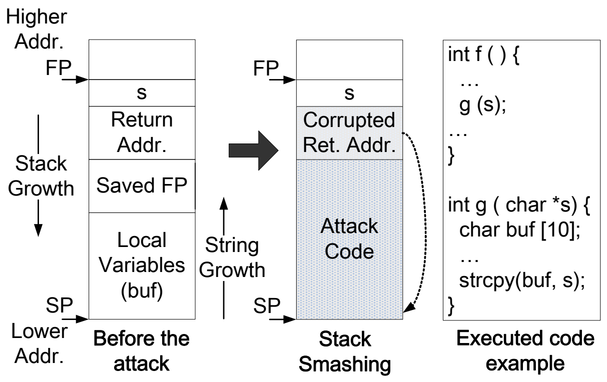

Fig. 1: Stack Smashing

completes. As a result, the program-execution control is finally hijacked by the injected attack code.

B. *Related Work*

The most straightforward way to solve the buffer-overflow attack is to prohibit the execution of codes stored in data segments. For example, AMD Athlon64 employs this protection. However, some programs attempt to generate an executable code at run time, e.g. just-in-time compiler.

So far, many techniques to address the buffer-overflow attacks have been proposed. They can be classified into two types: static and dynamic. The static approach generates a secure object code based on source code analysis. For example, the paper [10] formulates the detection of buffer overflows as an integer range analysis problem in order to find the potential of stack smashing. *SASI* introduced in [4] inserts reference-monitor codes into application programs to observe program-execution behavior. *StackGuard,* which is a patch to gcc, is another static approach to defending the stack smashing [3].

One of the main drawbacks of the static approach is code compatibility, because it requires a code translation or re-compilation. On the other hand, the dynamic approach does not have this negative effect. A dynamically loadable library (DLL) called *libsafe* is used to check vulnerabilities before un-trusted standard C libraries are executed [1]. Similarly, *libverify* is another DLL which injects a verification code at the beginning of the process execution via a binary re-writing. *StackGhost* exploits cleverly the register window of the SPARC architecture [5]. In such architecture, a return address is saved in the register window. Another approach called *SRAS* (*Secure Return Address Stack*) is an LIFO fashion small memory embedded in the microprocessor core [8]. SRAS is a straightforward but very efficient architectural support to prevent the buffer-overflow attacks. A return address is pushed onto not only the memory stack but also SRAS, and they are compared to detect stack smashing when the corresponding return instruction is executed.

Since SCache belongs to the dynamic approach, code compatibility can be maintained. In addition, we do not modify the library and OS. Unlike StackGhost, our approach is independent of the microarchitecture of processor. We exploit the random-access large cache to store the copied data. Therefore, SCache has enough capacity and can work well even if function calls are performed with non-LIFO fashion, e.g. *longjump()*. Another simple hardware support is to XOR the return-address value with a secret key, whereas SCache does not require any key information.

III. THE SCACHE ARCHITECTURE

A. *Main Idea*

In order to compensate for the poor performance of memory systems, a number of microprocessor chips employ on-chip caches. In such kind of systems, the return addresses are transferred to (or from) the memory stack through the on-chip caches. Therefore, if it is possible to protect the value of return address on the cache, we can prevent stack smashing without affecting the structure of complex microprocessor. In order to achieve such a hardware protection, SCache attempts to make one or more copies of the return address when it is stored into the cache. We call a cache line which includes the copy of the return address a *replica line*. Actually, the replica line is not a complete copy of whole cache line. **Only the return address is copied.** Since the replica line can be placed in the cache set indexed by the reference address of the current return-address store, we can generate at most "*Asso*-1" replica lines where *Asso* is the cache associativity. We prohibit any cache accesses except return-address stores to overwrite the replica lines. Therefore, the replica lines are treated as read only. When a return address is popped off from the memory stack, SCache selects one of the replica data and compares it with the popped original return address. If they are exactly the same, we can ensure that the popped return address is safe. Otherwise, it means that a return-address corruption takes place, thus a signal to report security status is sent to the processor in order to terminate the current program execution.

B. *Organization and Operation*

Fig. 2 shows the structure of a four-way set-associative SCache. Here, we assume that the number of replica lines to be generated for each return-address store (noted as *Nrep*) is two. Unlike conventional caches, one-bit replica flag (*R-flag*) is added to each tag entry. The R-flag is set to one if the corresponding cache line is a replica line, otherwise it is reset to zero. Moreover, a multiplexer to select a replica data and a 32-bit comparator for examining return-address corruption are required. Fig. 3 illustrates its operation on return-address load/store accesses. When a return address is pushed, i.e., on a return-address store, the cache works as follows.

W1. With the same manner as conventional caches, the set indexed by the reference address is accessed for tag checking. The corresponding R-flags are also read in parallel.

W2. Then the return address is stored into the cache-hit line. We call the target line *the master line*. Namely, the master line includes the original value of the return address.

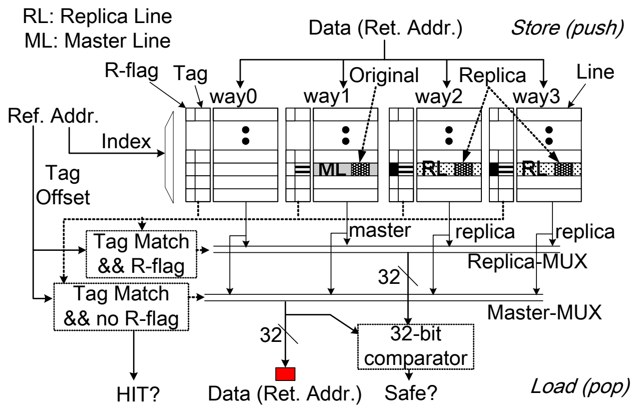

Fig. 2 : 4-way set-associative SCache

W3. If some replica lines already exist in the indexed set, the return address is stored into them in order to avoid a coherence problem. The replica lines can be detected by examining the tag and the R-flag at the step W1, because the corresponding replica lines have the same tag information and a valid R-flag.

W4. New replica lines are generated by writing the return-address value into other non-replica line(s) and setting the corresponding R-flag(s) to one. The tag and offset information of the original write operation performed at the step W2 is used to make the replica line(s).

Note that the above example assumes a cache hit. However, even on a miss, the cache works in the same manner after the line replacement is completed. SCache **does not make a complete copy of whole master line**, but just writes one word return-address value to the several lines in parallel. In other words, the contents of the replica lines are not exactly the same as those of the master line. In addition, a replica line can include several return addresses if the offset addresses are different. On the other hand, when a return-address load is executed, the cache examines whether or not the popped return address is secure as follows.

R1. The cache activates all the ways to read cache lines, tags, and R-flags. Then, the master line, which has the matching tag and the invalid R-flag, is selected. The return-address data read from the master line is sent to the processor.

R2. The replica lines having the matching tag are searched in the referenced set. If there are several replica lines corresponding to the current return-address load, one of them is randomly selected. Otherwise, the cache reports to the processor that we can not ensure the security of current return-address value.

R3. The return address read from the master line is compared with that obtained from the selected replica line. For this examination, only one replica data is selected and compared with the original one, because all of the replica lines include the same copy of the original return-address value. If the 32-bit comparison result is not a match, the cache alarms to terminate the program execution due to the insecure return-address load.

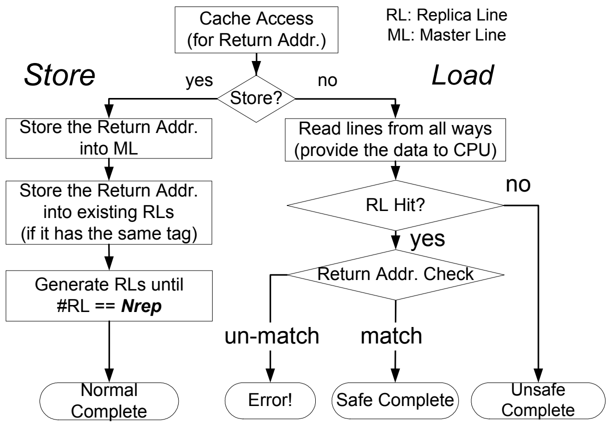
Cache Access (for Return Addr.)
RL: Replica Line
ML: Master Line
Store
Load
yes
Store?
no
Store the Return Addr. into ML
Read lines from all ways (provide the data to CPU)
Store the Return Addr. into existing RLs (if it has the same tag)
RL Hit?
no
yes
Return Addr. Check
Generate RLs until #RL == Nrep
un-match
match
Normal Complete
Error!
Safe Complete
Unsafe Complete

Fig. 3 : Operation on return-address accesses

In conventional write accesses, the tag checks and the data write (W1 and W2) are performed sequentially. This is because the state of cache can be updated, or store access can be started, after the confirmation that the target data exactly resides in the cache. Although SCache requires two more steps, W3 and W4, they can be executed in parallel with the step W2. Namely, the candidates of replica lines are determined based on the tag-check results performed in the step W1, and generating the replica lines can be done in parallel with the step W2. On the other hand, for read accesses, SCache completes to read the target data at the step R1 as well as conventional caches. In addition, the hardware components for SCache, a replica-line MUX and a 32-bit word comparator, do not affect the cache critical paths. Accordingly, SCache does not worsen the cache read-access time. Although SCache requires two more steps R2 and R3, the microprocessor does not need to wait the completion of these processes.

For normal accesses (i.e. non-return-address loads or stores), SCache operates as the same as conventional caches except that the **invalid** R-flag is included in the cache-hit condition. Therefore, the normal accesses can not modify the replica lines. For instance, let us consider a write operation to the next stack entry of the return address. We do not need to protect the write target entry. Here, we assume that the write target entry is allocated to the same cache line as the return-address value. In this scenario, the write operation is performed only to the master line, which has the invalid R-flag. Therefore, the associated replica-line which has the same tag information with that of the master line is not modified, and the write operation is completed successfully only to the master line.

The microprocessor needs to output a signal to indicate whether the current access targets a return address. This can be easily achieved by checking the source (or destination) operand of the current memory reference [8]. For many microprocessors, the return address is located to a special register, e.g. R31.

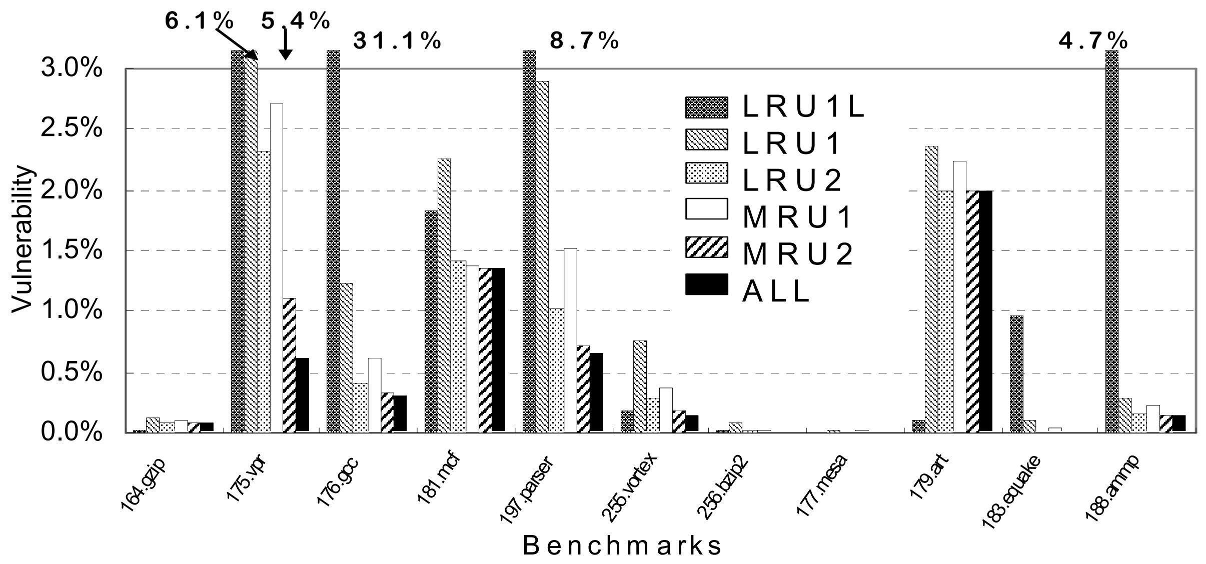

Fig. 4 : Vulnerability

IV. EVALUATION

In this section, we define six SCache models and discuss the ability of proposed approach. The SCache models differ in the number and the placement policy of replica lines.

A. Experimental Setup

We extended SimpleScalar tool set (ver.3.0d) [11] to support the SCache approach and executed seven integer programs and four floating-point programs from the SPEC2000 benchmark suite [12]. The small input data set was used to complete the whole program execution. In this evaluation, we assume that the L1 data cache size is 16 KB, the line size is 32 B, and the associativity is 4. Furthermore, a 4-way superscalar out-of-order execution is assumed. For other parameters, we used the simplescalar default value defined in [2]. In this section, we refer to the issued return-address load (or store) to the cache as the *IRA load* (or *store*). We use the following equation to evaluate the efficiency for security,

$$\textbf{Vulnerability} = (\textit{Nv-rald} / \textit{Nrald}) * 100,$$

where *Nrald* and *Nv-rald* are the total number of IRA loads in the program execution and that of insecure IRA loads (i.e. return-address loads without any replica line), respectively. On the other hand, to evaluate the energy overhead, we use the energy model,

$$Etotal = Erd + Ewt + Ewb + Emp,$$

where *Erd* and *Ewt* are the total energy consumed for cache read and write accesses, respectively. *Ewb* is another energy overhead caused by evicting dirty lines from the cache due to replica placement. *Emp* is the energy dissipated for cache-line replacements. We assume that the energy consumed for a next-level memory access is ten times larger than that for an L1 cache read access. Based on a 0.18 μm CMOS technology, we designed a 4KB SRAM array and estimated energy consumption. The circuits have been optimized to meet 3.0 ns access time. After the layout, we have measured energy consumption by performing Hspice circuit simulations with extracted load capacitances. First, we obtained the energy for accessing 1-bit memory cell that includes sensing and pre-charging. Next, we calculated the average energy for each operation based on the total number of bits to be accessed. Then we multiplied the energy by the number of events occurred during the program execution.

Since the cache associativity is assumed as four, we can generate at most three replica lines for each return-address store, i.e. *Nrep* = 3. Furthermore, there are two options for the replica line placement, LRU and MRU. Here, we define six SCache models: ***LRU1L***, ***LRU1***, ***LRU2***, ***MRU1***, ***MRU2***, and ***ALL***. LRU1 and LRU2 generate one and two replica line(s) with the LRU placement algorithm, respectively. MRU1 and MRU2 place replica line(s) on the MRU location except the master line. The ALL model makes the maximum number of replica lines. In these models, the replica lines are treated as the same as normal lines, i.e. they can be evicted from the cache. On the other hand, LRU1L prohibits evicting the replica lines, and they are released when the corresponding return-address load is issued to the cache. We compare the SCache models with a conventional low-power way-predicting cache, noted as ***CONV***. This model attempts to activate only the hit way which includes the reference data by employing an MRU-base way prediction [6][8], thereby saving cache-access energy. Here, we do not take the energy overhead caused by the way prediction, e.g. accessing an MRU table, into account. Note that the SCache models also perform the MRU-base way prediction. However, on each return-address load, they need to activate all the ways in spite of the correct way-prediction, as explained in Section II.

B. Vulnerability

Figure 4 shows the vulnerability of the SCache models. We should notice that conventional caches without any

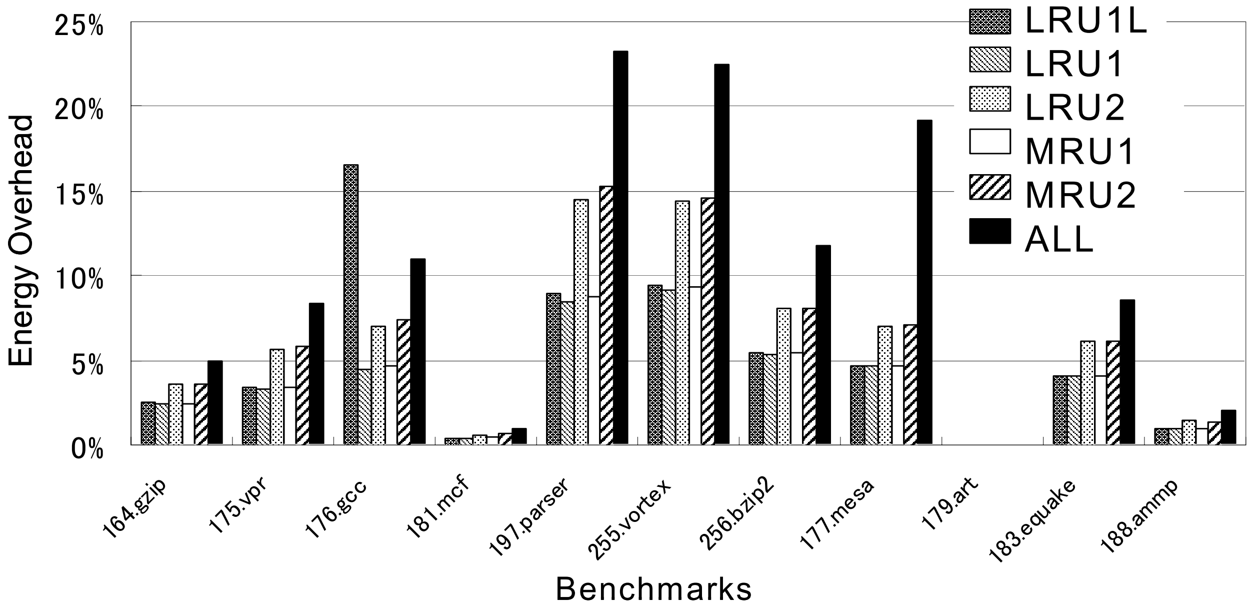

Fig. 5 : Energy Consumption

consideration for stack smashing have 100% of vulnerability.

First, we discuss the effects of the number of replica lines, *Nrep*. In this simulation, we assumed that the LRU policy is employed for the cache-line replacement on misses. Therefore, replica lines generated in LRU1 are easily evicted from the cache due to conflicts. As we expected, LRU2 produces better results than LRU1 for all benchmarks due to the increased number of replica lines. We see the same situation for the MRU based SCaches, MRU1 and MRU2. If they can perform a strict MRU placement, the number of replica lines does not affect the vulnerability. However, when a return address is stored into the cache, all existing replica lines with the same tag information are updated in order to avoid coherence problem, as explained in Section II. If a corresponding replica line already exists at the LRU location, MRU1 works as the same as LRU1. Therefore, MRU1 is more vulnerable than MRU2. For all but *181.mcf* and *179.art*, the most secure model ALL can protect more than 99.3% of IRA loads from stack smashing.

Next, we discuss the impact of the replica-line placement algorithm. The MRU strategy constantly achieves higher security than the LRU models if the number of replica lines to be generated at each return-address store is the same. This is because the MRU placement makes the replica lifetime longer as well as increasing the number of replica lines. However, against our expectation, LRU1L does not work well for some benchmarks. One of the reasons for this result is a hasty release of replica data caused by squished return-address loads.

C. Energy Overhead

Fig. 5 shows energy consumption for each benchmark program. All results are normalized to *CONV*. From the figure, we see that increasing the number of replica lines worsens energy efficiency. The ALL model increases energy consumption by about 23% in the worst case, *197.parser*.

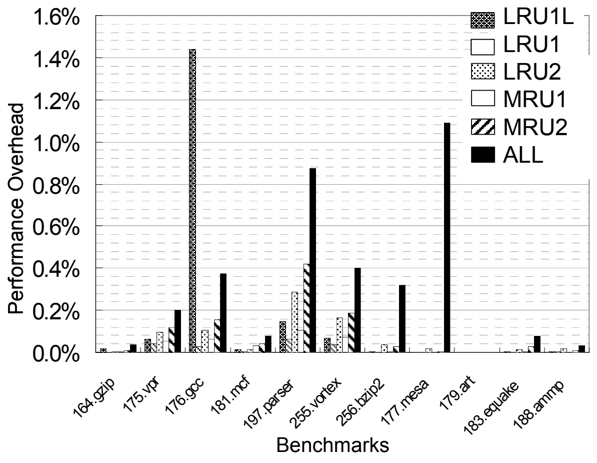

Fig. 6 : Performance

On the other hand, replica-line placement algorithm does not give a large impact on energy.

The SCache models have the same energy overhead for read accesses. Since all of the ways in SCache are activated on each IRA load, the read-access energy depends on not the number of replica lines generated but the total number of IRA loads. In contrast, the energy dissipated for write accesses *Ewt* increases with the increase in the number of replica lines to be generated. This situation can be seen for *Emp* due to the increased cache-miss rates. For instance, in case of 177.mesa, the ALL model worsens the cache-miss rate from 0.14% to 1.08%, thereby increasing the energy for cache-line replacements *Emp*. For the programs with small energy overhead, *181.mcf* and *179.art*, it is observed that the increase in *Erd*, *Ewt*, and *Emp* are trivial. For these benchmarks, the total energy is originally dominated by *Emp*, 51% for *181.mcf* and 62% for *179.art*, due to the higher cache-miss rates. Therefore, the energy overhead caused by the replica lines is relatively hidden.

D. Performance Overhead

We evaluate the impact of the SCache approach on processor performance. Increasing the number of replica lines, *Nrep*, worsens cache-hit rates, thus the processor performance will be degraded. This negative effect appears clearly on the MRU-based models. Fig. 6 reports the performance overhead caused by the SCache scheme. For the ALL model, the increase in execution time is at most 1.1%, *177.mesa*. Furthermore, we see that in many cases the performance overhead is less than 0.4%. Therefore, we believe that the performance degradation caused by SCache is negligible.

V. Conclusions

In this paper, we have presented a secure cache architecture, called SCache. The cache makes it possible to detect stack smashing at run time. The stack smashing alters a function return address for transferring the program-execution control to an injected malicious code. By making one or more replica lines in the large cache area, we can protect the return address. In this evaluation, we have estimated energy consumption based on an SRAM layout design which does not include peripheral circuits for SCache, e.g. a selector for replica lines, control logic, and so on. Our ongoing work is to design a complete SCache core. Another future work is to explore the SCache design space with various cache configurations, and to establish an optimization technique to find the best point to maximize the energy-security efficiency.

Acknowledgements

I would like to thank Prof. Shingi Tomita, Prof. Hiroto Yasuura, and all other members of PRESTO "information infrastructure and applications" research group for discussing at technical meetings. The VLSI chip in this study has been fabricated in the chip fabrication program of VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Hitachi Ltd. and Dai Nippon Printing Corporation. This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, and for Encouragement of Young Scientists (A), 17680005.

References

[1] A.Baratloo, N.Singh, and T.Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," *Proc. of 2000 USENIX Annual Technical Conference*, June 2000.

[2] D.Burger and T.M.Austin, "The SimpleScalar Tool Set, Vertion 2.0," *Univ. of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, June, 1997.

[3] 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.

[4] U.Erlingsson and F.B.Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. of the workshop on New security paradigm*, 1999.

[5] M.Frantzen and M.Shuey, "StackGhost: Hardware Facilitated Stack Protection," *Proc. of the 10th USENIX Security Symposium*, Aug. 2001.

[6] K.Inoue, T.Ishihara, and K.Murakami, "Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption," *Proc. of the Int. Symp.on Low Power Electronics and Design*, pp. 273--275, Aug. 1999.

[7] K. Inoue, "Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," *Workshop on Architectural Support for Security and Anti-Virus (WASSA04),* pp.77-85, Oct. 2004.

[8] R.B.Lee, D.K.Karig, J.P.McGregor, and Z.Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," *Proc. of the Int. Conf. on Security in Pervasive Computing,* Mar. 2003.

[9] M.D.Powell, A.Agarwal, T.N.Vijaykumar, B.Falsafi, and K.Roy, "Redicing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," *Proc. of the 34th Int. Symp. on Microarchitecture*, pp.54--65, Dec. 2001.

[10] 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.

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

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