Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache

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

https://doi.org/10.15017/6355

出版情報：IEEE International Conference on Electronics, Circuits and Systems, pp.1093-1096, 2006-12. IEEE
バージョン：
権利関係：

Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache

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*—This paper proposes an efficient cache line management algorithm for a security-aware cache architecture (SCache). SCache attempts to detect the corruption of return address values at runtime. When a return address store is executed, the cache generates a replica of the return address. This copied data is treated as read only. Subsequently, when the corresponding return address load is performed, the cache verifies the return address value loaded from the memory stack by means of comparing it with the replica data. Unfortunately, since the replica data is also a candidate for cache line replacements, SCache does not work well for application programs that cause higher cache miss rates. To resolve this issue, a lock and unlock data management algorithm is proposed in order to improve the security of SCache. The experimental results show that a proposed SCache model can protect about 99% of return address loads from the threat of buffer overflow attacks, while it worsens the processor performance by only 1%, compared with a non-secure conventional cache.**

I. INTRODUCTION

Buffer overflow vulnerability is a well known problem that exists in many computer systems. A number of malicious attacks have exploited this weakness, such as the Code Red worm of 2001 and Blaster in 2003. By exploiting the buffer overflow vulnerability, attackers attempt to change a return address value to refer to the start of a piece of injected malicious code. The microprocessor's program counter is set to the corrupted return address when execution of the called function has completed. As a result, program execution control is hijacked by the injected code.

In order to resolve the buffer overflow issue, a number of static or dynamic approaches have been proposed [1,2,3,4,5,6,7,8]. One of the main drawbacks of the static approach is code compatibility, because it requires code translation or recompilation. On the other hand, the dynamic approach does not have this problem [1][3][4]. *SRAS* (*Secure Return Address Stack*) is a LIFO-fashion small memory embedded in a microprocessor core [6]. SRAS is a straightforward but very efficient hardware defense against buffer overflow attacks. A return address is pushed not only onto the memory stack but also onto SRAS, and later the stored address is used to detect any corruption of the return address value when the corresponding return address load is performed. In [5] another architectural defense is proposed which extends the on-chip data cache in order to prevent buffer overflow attacks. This is called *SCache* (*Secure Cache*). Since cache memories support random accesses, the approach works well even if function calls and returns are performed in a non-LIFO fashion, e.g., *longjump()*. Also, SCache can be easily implemented, because it does not require the modification of complex processor cores.

In this paper a new cache line management algorithm is proposed to improve the security effectiveness of SCache. Unfortunately, SCache does not work effectively for application programs which cause a number of cache misses. This is because SCache attempts to create a copy of the return address when it is pushed onto the memory stack. The replicated data is kept in the cache, and later used to verify the integrity of the return address value. Therefore, if a number of cache line replacements take place, the replica data may easily be evicted from the cache. In this case, SCache cannot verify the return address value, resulting in false negatives. To resolve this issue, the "Lock and Unlock" algorithm is introduced to manage the replicated data. This approach improves the security effectiveness of SCache with only a trivial performance overhead.

This paper is organized as follows. Section II briefly explains SCache and discusses its limitations. Section III proposes a new cache line management algorithm to improve the security effectiveness of SCache. In Section 4, the performance and vulnerabilities of the proposed approach are evaluated. Finally, the paper is concluded in Section 5.

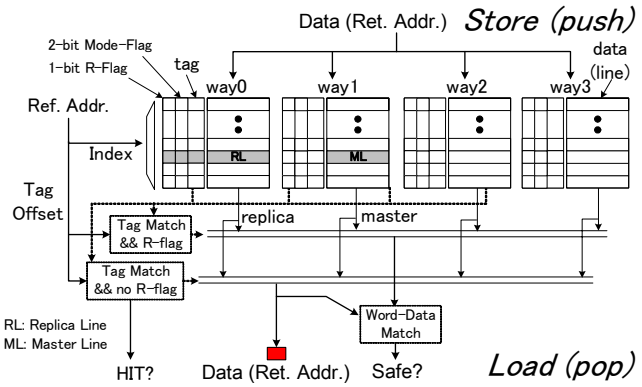

Figure 1: Four-way set associative SCache

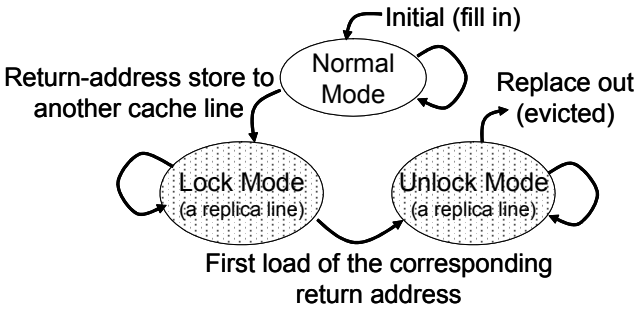

Figure 2: State transition diagram of cache lines

II. SCache Approach

A. SCache Configuration and Operation

In order to prevent the hijacking of program execution control, SCache detects return address corruption at runtime. On a return address store, the original return address value is written to the cache line. The line that contains the original return address value is called the *master line*. At the same time, SCache selects another line in the same cache set and stores the return address value in that line. Effectively, SCache creates a copy of the return address value in the cache. The cache line containing the copy of the return address value is called the *replica line*. The replica line has the same tag as that of the master line. SCache prohibits any accesses that overwrite the replica line, other than return address stores. Therefore, the replica line is treated as read only. When a return address is popped from the memory stack, SCache compares the popped data with the copy in the associated replica line. If they are the same, then the popped return address is safe. Otherwise, return address corruption has occurred. In this case, a signal reporting the security status is sent to the microprocessor in order to terminate execution of the current program. Reference [5] explains this in detail.

B. Limitations

SCache can detect return address corruption whenever an associated replica line exists in the cache. In other words, if a replica line cannot be found in the cache, the integrity of the return address value popped from the memory stack cannot be verified. When a cache miss takes place, SCache deals with replica lines like other cache lines, i.e., the replica lines may be evicted from the cache. Therefore, for application programs which frequently cause cache misses, SCache may not be able to detect return address corruption. The easiest way to resolve this issue is to prohibit the eviction of replica lines from the cache. In such a cache line locking mechanism, the replica line must be invalidated carefully, i.e., the cache area used by the replica line must be released at an appropriate time. This is because an early invalidation impairs security, whereas a late invalidation reduces the effective available cache capacity.

A simple approach is to invalidate the locked replica line when the corresponding return address load is issued. However, the possibility of early invalidation remains. A number of microprocessors employ various predictions to improve performance, e.g., branch prediction and value prediction. In such high-performance microprocessors, some instructions may be squashed when a prediction miss occurs. Therefore, the return address load to be squashed then prematurely invalidates the corresponding replica line.

III. Lock and Unlock Management

In order to overcome the limitations described in Section II, we propose a new cache line management algorithm, called *LUL* (*Lock and UnLock*). To implement the LUL method, the following three operation modes are supported for each cache line:

- **Normal mode:** the cache line works the same way as in a conventional cache. Namely, the cache line follows the cache line replacement algorithm employed, e.g., LRU (Least Recently Used). All of the non-replica lines are allocated to this mode.
- **Lock mode:** the eviction of the cache line in this mode is prohibited, i.e., the cache line continues to reside in the cache. Only replica lines can work in this mode.
- **Unlock mode:** the cache line works as a replica line, but unlike lock mode the line follows the original replacement algorithm, just like normal mode.

Figure 1 shows the configuration of a proposed four-way set associative SCache. Unlike the original SCache structure, a two-bit mode flag is implemented for each cache line in order to indicate the current operation mode. Figure 2 shows the mode transition diagram of a cache line. The proposed enhanced SCache works as follows:

- When a return address is stored in the cache, another line in the same cache set is assigned as a replica line. Then, a copy of the return address value

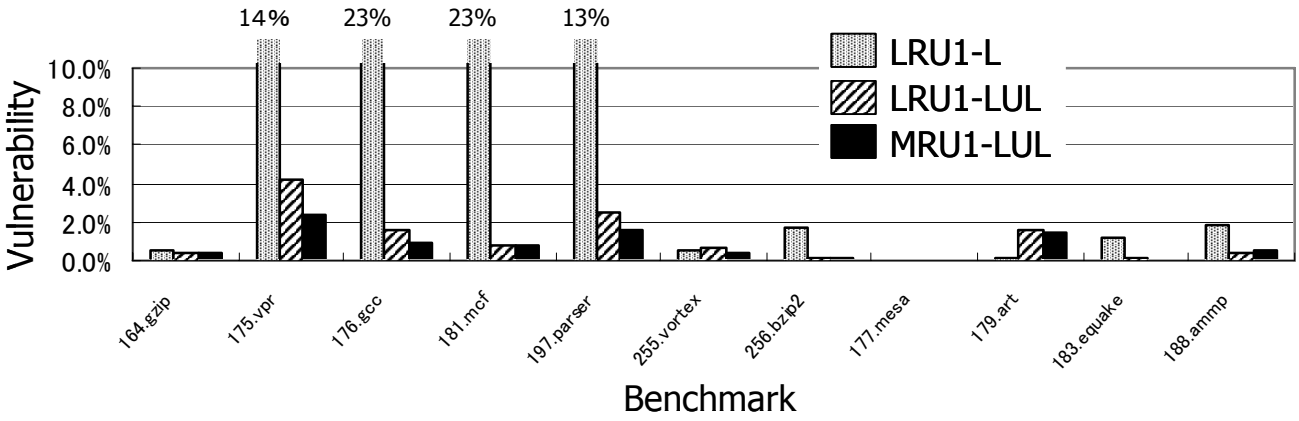

Figure 3: Vulnerability

is stored in the replica line, which then changes to lock mode. Eviction of the replica line is prohibited.

- When a return address load is executed, the associated replica line changes to unlock mode. Unlike the original SCache approach, the replica line can stay in the cache for a longer time. This avoids the early invalidation problem explained in Section II.

IV. Quantitative Evaluation

A. Experimental Set-Up

We extended the SimpleScalar toolset (ver. 3.0d) [9] to support the SCache approach and executed seven integer programs and four floating-point programs from the SPEC2000 benchmark suite [10]. The small input data set provided by the SPEC was used to complete the whole program execution. We assume a four-way out-of-order superscalar microprocessor as shown in Table 1. Here, we assume that the L1 data cache size is 16 KB, the line size is 32 B, and the associativity is 4. In this section, we refer to the issued return address load as the *IRA load*. We use the following equation to evaluate the security effectiveness:

Vulnerability = (Nv-rald / Nrald) * 100,

where *Nrald* and *Nv-rald* are the total number of IRA loads for the program execution, and that of insecure IRA loads (i.e., return address loads with no replica line), respectively. We compare the vulnerability and performance overhead for the following three modes:

- LRU1-L: This is an original SCache model that supports cache line locking. When a return address store is performed, this model generates a replica line in lock mode at the LRU location. When the return address load is executed, the associated replica line is invalidated.
- LRU1-LUL: This is a proposed SCache model which supports the LUL algorithm. When a return address store is issued to the cache, a replica line is generated at the LRU location.
- MRU1-LUL: This model is the same as the LRU1-LUL model, except that MRU1-LUL generates a replica line at the MRU location in the indexed cache set. Therefore, the generated replica line can stay in the cache longer than for LRU1-LUL.

Table 1: Processor Configuration

	4-way Out-Of-Order Superscalar
Fetch/Issue/Commit Width	4
Branch Prediction	Gshare(4K-entry), 4-way BTB (512-set), RAS(8-entry)
RUU	32
LSQ	16
L1 D-Cache	4-way 16KB, 32B lines
L1 I-Cache	2-way 32KB, 64B lines
L2 Cache	4-way 512KB, 64B lines
Memory Latency	64 cycles
Function Unit iALU/iMUL/fALU/fMUL	4/2/4/2

B. Vulnerability

Figure 3 shows the vulnerability of the three SCache models. Note that conventional caches without any provisions against stack smashing have 100% vulnerability. For all but four of the benchmark programs, *175.vpr, 176.gcc, 181.mcf,* and *197.parser*, LRU1-L can achieve less than 2% vulnerability, i.e., 98% of return address loads can be protected. However, for the four benchmarks mentioned, the vulnerability factors are very high. This is because for these programs, we see higher branch prediction miss rates,

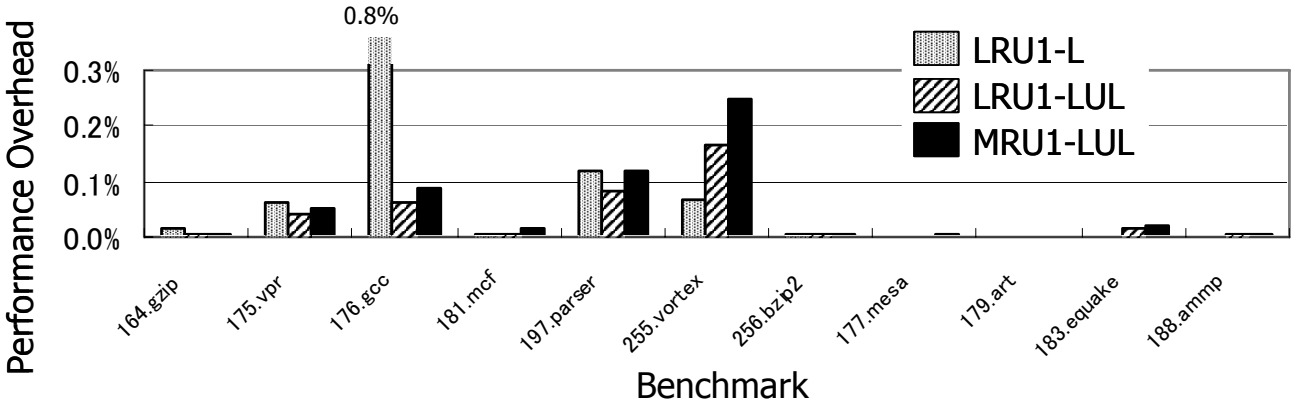

Figure 4: Performance Overhead

causing the early invalidation problem. On the other hand, our LUL strategy reduces the vulnerability very effectively for these programs. For almost all the programs, we can achieve less than 2% vulnerability. Notably, the MRU1-LUL model produces better results than the LRU1-LUL model. This is because the MRU replica placement strategy makes it difficult to evict replica lines from the cache. On average (using geometric mean), LRU1-LUL and MRU1-LUL achieve less than 0.38% and 0.41% vulnerability respectively. In other words, the proposed enhanced SCache models can be shown to accurately detect return address corruption.

C. *Performance Overhead*

Figure 4 shows the performance overhead caused by the SCache models. Although the LUL strategy can effectively avoid the eviction of replica lines, this feature increases the cache miss rate due to conflicts between normal and replica lines. However, as we can see in the figure, the performance degradation is at most 0.25%, for *255.vortex* with MRU1-LUL. Notably, for five benchmarks, *256.bzip2, 177.mesa, 179.art, 183.equake,* and *188.ammp*, the performance is effectively the same as a microprocessor which has no security provisions. Therefore, we believe that the performance degradation caused by SCache is negligible.

V. CONCLUSIONS

In this paper, in order to improve the security effectiveness of SCache, a cache line management technique called LUL (Lock and UnLock algorithm) was proposed. In this approach, the replica lines used to verify the integrity of return address values are locked in the cache. The replica lines subsequently follow the conventional line replacement algorithm when a corresponding return address load is executed.

In the evaluation, it was observed that a proposed enhanced SCache model, i.e., MRU1-LUL, can detect buffer overflow attacks for approximately 99% of return address loads, whereas the performance overhead is less than 1%. This means that the approach can enable secure computing without adversely affecting microprocessor performance. Hardware level protection is believed to be very important in the protection of computer systems. Ongoing work will consider other hardware techniques to prevent malicious attacks, and develop a platform for secure computing.

ACKNOWLEDGMENTS

This research was supported in part by the PREST, 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] M. L. Corliss, E. C. Lewis, and A. Roth, “Using DISE to Protect Return Addresses from Attack,” *Proc. of the international Workshop on Architectural Support for Security and Anti-Virus,* pp.61-68, Oct. 2004.

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

[5] K. Inoue, ``Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," Proc. of the International Workshop on Architectural Support for Security and Anti-Virus, pp.77-85, Oct. 2004.

[6] 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.

[7] 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.

[8] D. Ye and D. Kaeli, ``A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing,” *Proc. of the international Workshop on Architectural Support for Security and Anti-Virus,* pp.69-76, Oct. 2004.

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

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