IEICE TRANS. INF. & SYST., VOL.E93–D, NO.5 MAY 2010

PAPER Special Section on Information and Communication System Security

Eliminating Cell Broadband Engine™ DMA Buffer Overflows

Masana MURASE[†a)], ***Member***

SUMMARY This paper presents effective and efficient implementation techniques for DMA buffer overflow elimination on the Cell Broadband Engine™ (Cell/B.E.) processor. In the Cell/B.E. programming model, application developers manually issue DMA commands to transfer data from the system memory to the local memories of the Cell/B.E. cores. Although this allows us to eliminate cache misses or cache invalidation overhead, it requires careful management of the buffer arrays for DMA in the application programs to prevent DMA buffer overflows. To guard against DMA buffer overflows, we introduced safe DMA handling functions for the applications to use. To improve and minimize the performance overhead of buffer overflow prevention, we used three different optimization techniques that take advantage of SIMD operations: branch-hint-based optimizations, jump-table-based optimizations and self-modifying-based optimizations. Our optimized implementation prevents all DMA buffer overflows with minimal performance overhead, only 2.93% average slowdown in comparison to code without the buffer overflow protection.

key words: *DMA buffer overflow, DMA buffer overrun, Cell Broadband Engine™*

1. Introduction

The buffer overflow or overrun attack has been a computer security headache even before the famous Morris worm appeared in the 1980s. This problematic vulnerability comes from the lack of a data integrity model in the C and C++ programming languages. Due to the lack of automatic bounds-checking in C programs, unexpectedly large inputs can overflow buffer arrays and corrupt the stack or heap areas. In the worst case, a system can be taken over by inserting malicious code with a buffer-overflow attack.

To cope with this problem, several countermeasures have been proposed and implemented. One of the solutions is to develop applications with programming languages that support run-time bounds checking, such as Java. Such programming languages can stop the application's execution when they detect overflows or underflows of buffer arrays. Safe compiler technologies [1], [2] allow us to generate a safe executable that detects overflows without modifying the source. For example, it is possible to detect stack smashing buffer overflows by placing a 'canary' word before the return address on the stack. In the function prologue of the callee, a random 'canary' is stored and the canary is verified by the function epilogue before returning to the caller. PaX [3] or Exec Shield [4] use a different approach to protect against buffer overflows. Their approach prevents arbitrary malicious code execution by using memory protection mechanisms in the hardware or the operating system instead of detecting buffer overflows.

Such countermeasures are effective in commodity PC systems, but it is hard to apply them to specialized processors like the Cell Broadband Engine™ (Cell/B.E.) processor [5] because of differences in the processor architectures. The Cell/B.E. processor is a heterogeneous multi-core processor with 9 cores: one PowerPC Processor Element (PPE) and 8 Synergistic Processor Elements (SPEs). It is possible to apply the prior work against for PPE-side buffer overflows, because the PPE is a PowerPC-compatible chip and hosts a traditional software stack where the operating system or the hypervisor manages applications. On the SPE side, however, there are no system-level protection mechanisms against buffer overflows caused by an SPE program.

One of the special programming features of the Cell/B.E. processor is that users are required to manually copy data between the system memory and the local store (LS) attached to each SPE. This is because programmer-managed data copies can avoid cache misses and cache invalidation overhead, which enhances the application performance. Application programmers implement SPE code that explicitly issues direct memory access (DMA) commands to access the system memory.

From the security perspective, however, this Cell/B.E.-specific programming requirement compels application developers to carefully manage the DMA buffer arrays to avoid buffer overflows. In the worst case, DMA buffer overflows by the SPE could also overwrite the original code and data segments, which could allow arbitrary code execution. If that SPE application were running with the system root privilege, it would be possible for an adversary to hijack not only that SPE, but also the entire Cell/B.E. system. For example, once an SPE program with the root privilege is compromised, that hijacked SPE program can illegally access a file system or inject arbitrary malicious code on any DRAM areas via DMA, even if the PPE-side programs including the operating system or hypervisor are protected against buffer overrun attacks by the previous countermeasures. Unfortunately, in the current Cell/B.E. architecture, the SPE has no hardware memory protection even though it has this DMA buffer overflow vulnerability. It is hard to protect the original SPE code, stack, and data segments and to prevent arbitrary code execution. There is a need for DMA buffer overflow prevention to avoid malicious code injection.

Manuscript received July 30, 2009.
Manuscript revised December 17, 2009.

[†]The author is with IBM Research - Tokyo, IBM Japan Ltd., Yamato-shi, 242–8502 Japan.
a) E-mail: mmasana@jp.ibm.com

DOI: 10.1587/transinf.E93.D.1062

This paper addresses this DMA buffer overflow problem on the Cell/B.E. architecture, and introduces efficient and effective buffer overflow prevention techniques. Millions of Cell/B.E. chips are already deployed in game systems [6], embedded systems [7], and supercomputers [8], [9]. We are contributing to addressing the DMA security threats in a large variety of applications. The difficulties in realizing efficient and effective DMA buffer overflow prevention for the SPE are due to the following characteristics:

- Limited Resources
  - An SPE has only 256 KB of LS. Every SPE application must manage this small memory to allocate the code, stack, data, and heap areas. The DMA buffer overflow prevention code must not waste memory space.
- Lack of Memory Protection Mechanisms
  - An SPE has no hardware-level memory protection mechanisms such as ring-protection or an NX bit. Stack or heap buffer overflows can overwrite any of the content in the LS. Both kinds of overflows must be prevented before the DMA commands are executed.
- Minimum Performance Overhead
  - Most SPE applications are carefully optimized to accelerate computationally intensive tasks. The DMA buffer overflow prevention code must not impose noticeable performance overhead.

We have implemented optimized safe DMA library calls that replace the original DMA library calls. Our safe DMA library calls prevent buffer overflows from the stack and in the heap before the DMA starts to write to the buffer arrays. We optimized the safe DMA library calls by eliminating branch operations. The empirical results showed that we could prevent buffer overflows with minimal overhead, an average performance decline of 2.93% in comparison with the original and unmodified DMA library calls in the case of single-block DMA transfers.

The remainder of the paper is structured as follows. Section 2 discusses the related work and clarifies the drawbacks of each technology. Section 3 briefly reviews the Cell/B.E. processor architecture and then describes mechanisms for DMA buffer overflows. Section 4 focuses on the DMA buffer overflow problem and proposes several optimization methods. We evaluate our optimized buffer overflow elimination code in terms of the performance overhead in Sect. 5. Finally, Sect. 6 concludes this paper.

2. Previous Work

A number of attempts to address buffer overflow and overrun attacks have been made in the last few decades. Theoretically, it is possible to eliminate the risks of buffer overflows by performing bounds checking every time a program issues any write operation to a buffer array. In this section, we categorize the previous work into four groups and clarify the drawbacks in each approach.

2.1 Overflow-Free Approach

An overflow-free approach guarantees that buffer overflows and underflows are prevented at the programming language level. Java and .NET support this capability by performing bounds-checking in the runtime environment, in the Java Virtual Machine, or in .NET's Common Language Runtime. With this approach, application programmers do not need to worry about programming mistakes causing buffer overflows and underflows. Therefore, it is easy to implement buffer-overrun-free code in such languages.

Unfortunately, the Cell/B.E. programming environment, and especially SPE programming environment, supports native code execution only. The reason is that safe language environments impose overhead. In the Cell/B.E. processor, each core has only 256 KB of local memory in which code, data, and stack areas are allocated. It is impractical to implement safe language runtime systems supporting bounds-checking with such limited hardware resources. Also, the essence of the Cell/B.E. is to directly use the Cell/B.E. hardware acceleration features such as user-managed DMA with double-buffering, SIMD, and dual-instruction issues to enhance the application performance. Therefore, a C or C++ type of programming model is most suitable for the Cell/B.E. environment for the sake of performance and resource efficiency. We need a lightweight method to handle the DMA buffer overflows for the Cell/B.E. architecture.

2.2 Overflow Prevention Approach

An overflow prevention approach aims to detect buffer overflows before trying to execute any damaged code. Libsafe [10] is a dynamically loadable library that intercepts insecure library calls from the target applications, especially from insecure functions such as `strcpy()`, `gets()`, and `sprintf()`. For example, when a user program calls `strcpy()`, Libsafe's `strcpy()` is invoked before the libc original `strcpy()` is executed. First, Libsafe determines and compares the input string lengths and the upper bounds of the output buffers on the stack. If an input string is larger than its output buffer, Libsafe terminates that user program. Otherwise, it passes control to the libc original `strcpy()`. Since Libsafe is automatically and dynamically preloaded by the Linux library-preloading facility, users need not recompile their applications.

Libparanoia [11] also focuses on making unsafe libc functions secure. Libparanoia provides secure versions of the libc string manipulation functions with the same semantics and functions. When a program statically or dynamically linked to libparanoia calls `strcpy()`, libparanoia's `strcpy()` is invoked first, as in Libsafe. Libparanoia's `strcpy()` uses three steps. First, before copying strings,

libparanoia saves the frame pointer and return address in the data segment. Then it copies the strings from the input buffers into the local output buffers on the stack. Finally, libparanoia compares the current frame pointer and return addresses with the stored values. If those values do not match, libparanoia terminates the program. Libparanoia can detect buffer overflows only after the buffer overflows occur, whereas Libsafe can check for buffer overflows before invoking the insecure libc functions.

Libsafe and libparanoia focus only on stack smashing attacks. In contrast, our buffer elimination techniques deal with both stack buffer overflows and heap buffer overflows in an unified way.

2.3 Overflow Detection Approach

StackGuard [1], an overflow detection approach, inserts code to generate a random value (a 'canary') in the function prologue. The canary is placed next to the return address. If a buffer overflow occurs, both the canary and the return address will be replaced. Thus, it is possible to detect buffer overflows by checking if the canary has been changed. To implement this test, StackGuard also inserts canary checking code in the function epilogue. When the function exits, the canary checking code compares the current canary on the stack and the original one generated in the function prologue. If the canary has changed, then StackGuard terminates the program.

The main idea of the Stack Smashing Protect (SSP) [2] system is similar to that of StackGuard. In SSP, a more advanced method is implemented to deal with argument overwriting and frame pointer overwriting. SSP protects against such attacks by changing the stack layout. It places the local variables at the low end of all arrays, and inserts a canary (SSP calls the canary a 'guard') between the previous frame pointer and the array. In this layout, local variables are always unaffected by overflows of the arrays. This protects functions from argument overwriting. A function only needs to make a copy of the pointer arguments to local variables and use copies of these variables. In addition, since a guard is located at the upper bound of any array, we can detect any overwriting of the previous frame pointer by checking the guard before and after the use of arrays.

StackGuard and SSP focus on buffer overflows in the stack. In contrast, this paper addresses DMA buffer overflows not only in the stack, but also in the heap. Also, there are other problematic behaviors of SPE DMA buffer overflows. A DMA transfer request beyond the end of the LS can overwrite the original code and data segments as well as the application stack. This is because an operation such as a 'load' or 'store' to an address beyond the LS wraps to an address at the low end of the LS [12]. This means that an adversary could overwrite and disable the original code containing any stack inspection code by exploiting this address wrapping operation.

SPE application compilers like spu-gcc [13] and spuxlc [14] also provide options (`-fstack-check` or `-qcheck=stack`) to generate stack inspection code. Instead of generating a canary word, the generated inspection code tests for buffer overflows by checking the Available Stack Space stored in the stack pointer register [15]. The Available Stack Space word contains the limit of the free space for stack growth. This word is decremented when a new stack frame is allocated, and incremented when the top stack frame is released. The Available Stack Space word should always be positive or zero in normal cases. The stack inspection code inserted in the function prologue and epilogue can detect stack overflows if that word becomes negative.

However, the stack inspection code generated by SPE application compilers fails to detect buffer overflows, if the Available Stack Space word is positive, but incorrect. Unfortunately, the SPE cannot prevent the code area from being overwritten for the same reason as in the StackGuard and SSP cases.

2.4 Overrun Blocking Approach

PaX [3], Exec Shield [4], W^X [16], XD bit [17], and Enhanced Virus Protection [18] are categorized into this group. PaX, W^X, and Exec Shield are software-based solutions, while the XD bit and Enhanced Virus Protection are implemented in Intel chips and AMD chips, respectively. This general approach prevents the execution of injected malicious code by controlling the permission of memory pages or segments instead of preventing buffer overflows.

Address Space Layout Randomization (ASLR) in PaX, W^X, and Exec Shield is one of the techniques used to prevent illegal code execution, in which the base positions of the executable, the heap, and the stack are randomized when the application is loaded. This makes it harder to find a point of attack. Another way to prevent illegal code execution is to make the stack non-executable by using the IA-32 segmentation mechanism. As implemented in Exec Shield, the code segment size limit is set to the boundary of the application stack area and the application code area, which makes the code executable only within the application code area.

However, these methods cannot be applied to the Cell/B.E. architecture. First, the SPE does not support any hardware security mechanism for memory pages or segments. This means that the SPE does not have permission control for its local memory. Second, it is hard to apply the ASLR as used in PaX, W^X, or Exec Shield to the SPE applications, because randomizing the base address of the code, stack, and heap segments in the SPE executable wastes the limited local memory. We need to make efficient use of the limited resources.

3. DMA Buffer Overflows

3.1 Cell/B.E. Overview

Figure 1 is a block diagram of a Cell/B.E. processor. The Cell/B.E. processor has a heterogeneous multiple core architecture, in contrast to commercial PCs and servers, which

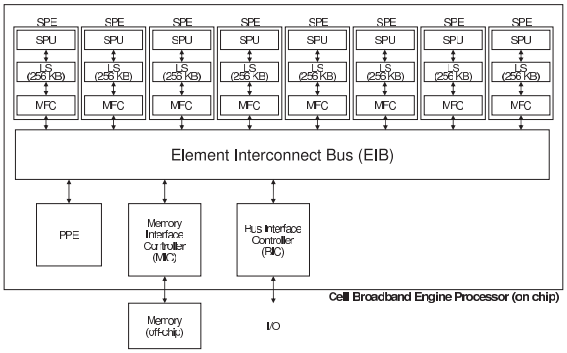


Fig. 1 Block diagram of the Cell Broadband Engine architecture.

typically use the homogeneous multi-core processor approach.

There are two kinds of cores on the chip: one 64-bit Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE is a PowerPC processor that runs an operating system to manage the computing resources such as processing units, system memory, and other peripherals. In contrast, the SPEs are computational workhorses that support RISC-style single instruction, multiple data (SIMD) computation, wide and large (128 128-bit) register files, and 256 KB of physical private memory, called the Local Store (LS). The LS is not a hardware cache, but a working memory area. Software on the SPEs or the PPE explicitly issues DMA commands to transfer data between the system memory and the LS. The high-bandwidth Element Interconnect Bus (EIB) connects these processor cores to each other and to the off-chip system memory and I/O.

3.2 DMA Transfer Types and APIs

The Cell/B.E. processor supports two types of DMA transfers at the hardware level: a single block DMA transfer and a list DMA transfer. The single block DMA transfer copies up to 16 KB of data per request between the system memory and the LS. In contrast, the list DMA transfer supports scatter-gather data copies. Since this paper addresses DMA buffer overflows on the SPE side, the DMA transfers from the system memory to the LS are focused on in this section.

Table 1 shows the DMA transfer APIs which are most frequently used. The `mfc_get()` call is used to perform a single block DMA transfer from the system memory to the LS. The `mfc_get()` call requires 6 parameters, of which the first 3 parameters are important here. The `ls` is a pointer to the destination DMA buffer array in the LS, `ea` is the effective address[†] of the data source, and the `size` is the data size. The `mfc_get()` call transfers a block of consecutive data of `size` bytes from `ea` to `ls`.

The `mfc_getl()` is used for a list DMA transfer, and requires 7 parameters. Compared with `mfc_get()`, `mfc_getl()` has no parameters for DMA transfer size. Instead, the third parameter, `list` and the fourth parameter, `list_size` are specific to this function. In the list DMA transfer, each `list` element defines the DMA transfer size and the offset from the `ea`. As specified by the `list` elements, `mfc_getl()` gathers and copies the scattered data from the effective addresses to the `ls`.

Table 1 APIs for SPE-initiated DMA calls that move data from an effective address to the LS.

`(void) mfc_get(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, uint32_t tid, uint32_t rid)`	
`ls`	destination LS address
`ea`	source effective address
`size`	DMA transfer size
`tag`	DMA tag ID
`tid`	transfer class ID
`rid`	replacement class ID
`(void) mfc_getl(volatile void *ls, uint64_t ea, mfc_list_element *list, uint32_t list_size, uint32_t tag, uint32_t tid, uint32_t rid)`	
`ls`	destination LS address
`ea`	source effective address
`list`	DMA list
`list_size`	DMA list size
`tag`	DMA tag ID
`tid`	transfer class ID
`rid`	replacement class ID

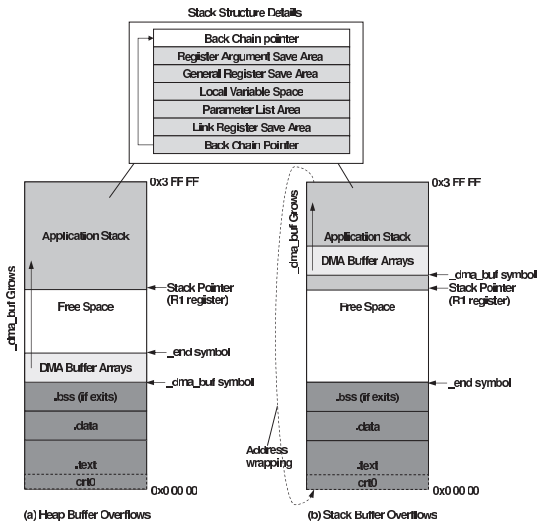


Fig. 2 Examples of DMA buffer overflows: (a) Stack smashing by illegal heap expansion, (b) Stack smashing by illegal local variable expansion.

3.3 DMA Buffer Overrun Attacks

Unfortunately, an adversary can easily execute arbitrary malicious code on an SPE by abusing the DMA buffer overflow vulnerabilities in the current Cell/B.E. runtime environment. The SPE has no memory protection mechanism such as page or segment permission control to protect against illegal access to its LS. Thus, an illegal heap expansion as well as an illegal expansion of the local variables could easily corrupt the SPE application stack (Fig. 2). In the worst case, an adversary might overwrite the link register save area holding a return address and jump to arbitrary malicious code.

[†]The effective address space is a 64-bit address space in the POWER architecture.

In the current Cell/B.E. architecture, a DMA transfer request beyond the size of the LS can overwrite the original code and data segment as well as the application stack. This is because an operation such as 'load' or 'store' to an address that is beyond the end of the LS is performed at the wrapped address. This means that the adversary could disable the original code containing buffer-overflow inspection code with malicious use of this address wrapping operation.

4. DMA Overflow Prevention

The current Cell/B.E. runtime system lacks DMA buffer overflow protection. To address this problem, this paper proposes an efficient and effective DMA overflow elimination technique to protect against both stack and heap buffer overflows. Our proposed idea would be categorized as an "Overflow Prevention Approach" in Sect. 2. First, our basic DMA overflow elimination mechanism is described. Then several optimization techniques to reduce the performance overhead of the inspection code are introduced.

4.1 Basic Idea

Considering the characteristics of DMA buffer overflows in the SPE, it is imperative that countermeasures against this problem prevent SPE applications from causing buffer overflows and terminate such applications before any attempts succeed.

Similar to the previous approaches of Libsafe and libparanoia, we replace the unsafe `mfc_get()` and `mfc_getl()` with safe versions. This is because we want minimal performance overhead and a small footprint for the inspection code.

Figure 3 is the pseudocode for our safe DMA function calls. Our `safe_mfc_get()` (Fig. 3 (a)), a safe version of a single block DMA transfer function, obtains the current stack pointer and the backchain pointer in Lines 2 and 3. In Line 4, it computes the upper bound of the allocated buffers in the LS with the specified `size`. In Line 5, `safe_mfc_get()` compares `ls` and `stack` to check if `ls` is or is not located in the stack. If `ls` is defined as a local variable in the stack, our safe `safe_mfc_get()` tests whether or not the upper bound of the allocated buffers exceeds the latest backchain pointer in Line 6. It immediately terminates the SPE application when it detects that `upper_bound` might smash into the wall of the `bcp` in Line 7. If `ls` is located in the heap area or in the data area, then `safe_mfc_get()` performs the conditional check for the `upper_bound` in Line 10. Similar to Line 6, it checks if the `upper_bound` is greater than or equal to the current stack pointer, instead of using the latest backchain pointer to protect the stack. If there are no violations for the use of the DMA buffer arrays, then `safe_mfc_get()` issues a DMA GET request to copy the data from the system memory into the LS.

We use the same approach as `safe_mfc_get()` for `safe_mfc_getl()` to eliminate buffer overflows. The additional operations in `safe_mfc_getl()` are computing the upper bounds of the target buffer arrays by parsing `list`. Looking back at the original interface of `mfc_getl()` again, there is no information about the total DMA transfer size in the parameters. To obtain the size of the total DMA transfer, it is necessary to extract DMA transfer size from each DMA list element. We perform these operations in Lines 4 to 7 in Fig. 3 (b). Once a DMA list GET request is issued, the SPE's DMA hardware processes a scatter/gather operation specialized for list DMA copies, and locates the gathered data in the `ls`. For this reason, the upper bounds checking including the calculation of the total transferred data size must be performed before issuing any DMA list GET request as shown in Fig. 3 (b).

Although, in the current implementation, application programmers must explicitly specify which versions of the DMA functions are used in the source code, this switching feature could be implemented as a compiler option. Methods of secure DMA code insertion using compilers are possible future work.

```
1  safe_mfc_get(ls, ea, size, tag, tid, rid) {
2    stack:=current stack pointer
3    bcp:=latest backchain pointer
4    upper_bound:=ls+size
5    if (ls>stack) then
6      if (upper_bound>=bcp) then
7        halt with an error
8      end if
9    else
10     if (upper_bound>=stack) then
11       halt with an error
12     end if
13   end if
14   issue the dma GET request
15 }
```

(a) Pseudocode of safe_mfc_get()

```
1  safe_mfc_getl(ls,ea,list,list_size,tag,tid,rid) {
2    stack:=current stack pointer
3    bcp:=latest backchain pointer
4    for i:=0 to (list_size >> 3) do
5    begin
6      {size:=size+list[i].size}
7    end
8    upper_bound:=ls+size;
9    if (ls>stack) then
10     if (upper_bound>=bcp) then
11       halt with an error
12     end if
13   else
14     if (upper_bound>=stack) then
15       halt with an error
16     end if
17   end if
18   issue a dma list GET request
19 }
```

(b) Pseudocode of safe_mfc_getl()

Fig. 3 The code examples of DMA buffer overflow elimination.

4.2 Code Optimization

In Fig. 3, there are many branches in both pieces of code. In SPE code, branch operations are relatively expensive. In the worst case, the branch penalty is 19 clock cycles. This is because the SPE has no branch prediction circuit. In general, most SPE applications slice a large amount of data into small pieces to fit in the LS. They transfer and process each small chunk of data in the loop until the final chunk arrives. Thus, this branch penalty would cumulatively in-

crease. Thus, branch elimination or branch penalty reduction is the primary optimization target. This section focuses on the single block DMA, but the same ideas can also be used for the list DMA.

In the current work, we have implemented three different methods to optimize branches in our original buffer overflow elimination. The first approach is an intuitive one of giving the compiler branch hints so the software can speculate that the code will branch to the target path. We call this approach branch-hint-based optimization. The second approach is a jump-table based approach. In general, a jump table is implemented as a function pointer array, and the code branches to a target path by choosing the index of that array and calling the selected function pointer. We used SIMD operations for the table content selection. The third approach is an unique approach. The inspection code dynamically overwrites the instructions according to their branch paths. We call this approach self-modifying-based optimization. The details of those approaches are described below.

Figure 4 (a) gives pseudocode for the first approach, `safe_mfc_get_A()`. To eliminate the nested branches in the original `safe_mfc_get()`, this new approach uses `spu_cmpgt` and `spu_sel`, which are both SPU SIMD operations. In Line 5, `safe_mfc_get_A()` compares `stack` and `ls`, and stores the result of the comparison into `cmp_r1`. When `ls` is located in the stack, which means that `ls` is larger than `stack`, all of the bits of `cmp_r1` become zeros. Otherwise, `cmp_r1` is full of ones. In Lines 6–7, the code checks if the `upper_bound` is greater than `stack` or `bcp`, which corresponds to Lines 6 and 10 in Fig. 3 (a). The `spu_sel(a, b, c)` is a key SIMD operation, where the first parameter, `a` or the second parameter, `b` is selected based on the third parameter `c`. The first `spu_sel` in Line 8 checks if `ls` is allocated in the stack or in the heap, and the second `spu_sel` in Line 9 checks for stack buffer overflows or heap violations following the results of the first `spu_sel` operation. Finally, it branches to a target path depending on the result of the second `spu_sel` operation. In this branch operation, we have added the `__builtin_expect` directive for a branch hint to the compiler. The code predicts that the conditional statement is false, which does not cause any performance penalty in the normal case. If the code detects a buffer overflow (from an incorrect prediction), then it terminates the application.

```
1  safe_mfc_get_A(ls, ea, size, tag, tid, rid) {
2   vector stack:=current stack pointer
3   vector bcp:=latest backchain pointer
4   vector upper_bound:=spu_add(ls, size)
5   vector cmp_r1:=spu_cmpgt(stack, ls)
6   vector cmp_r2:=spu_cmpgt(upper_bound, stack)
7   vector cmp_r3:=spu_cmpgt(upper_bound, bcp);
8   vector mask1:=spu_sel(cmp_r3, cmp_r2, cmp_r1);
9   vector results:=spu_sel(all0, all1, mask1);
10  if (__builtin_expect(results != 0, 0)) then
11   halt
12  endif
13  issue a dma GET request
14 }
```

(a) Branch-hint based optimization

```
1 static vector op1:={nop operations}
2 static vector op2:={halt operations}
3 safe_mfc_get_B(ls, ea, size, tag, tid, rid) {
4  snipped // same as line 2-8 of safe_mfc_get_A
5  /* second spu_sel */
6  func_t op:=(func_t)spu_sel(op1, op2, mask1);
7  op();
8  issue a dma GET request
9 }
```

(b) Jump-table based branch elimination

```
1  static vector op1:={nop operations}
2  static vector op2:={halt operations}
3  safe_mfc_get_C(ls, ea, size, tag, tid, rid) {
4   snipped // same as line 2-8 of safe_mfc_get_A
5   INLINE_ASM(
6    /* second spu_sel */
7    select op, op1, op2, mask1
8    store op, label_1
9    /* some code consuming 6 cycles here */
10   label_1:
11    nop
12    lnop
13    nop
14    lnop
15   );
16  issue a dma GET request
17 }
```

(c) Self-modifying based branch elimination

Fig. 4 The pseudo code for three optimization methods.

Figure 4 (b) shows the second approach, which does not use the `__builtin_expect` directive. To completely eliminate branches, `safe_mfc_get_B()` causes the second `spu_sel` in Line 6 to return a value that can be cast as a function pointer. If this DMA function call might cause a buffer overflow, then the `spu_sel` returns a value for a 'halt' instruction. Otherwise, it returns a value for a 'nop' instruction. Then `safe_mfc_get_B()` invokes `f()`, which is the result of the second `spu_sel`. However, this approach has some overhead for the function call using a function pointer.

The pseudocode in Fig. 4 (c) is a special implementation to completely avoid branches. As the description implies, `safe_mfc_get_C()` dynamically overwrites its own code to branch to the target path. Our code protects against overwriting the .text segment of an SPE application by using DMA buffer overflows. This special implementation can be safe. The difference between `safe_mfc_get_B()` and `safe_mfc_get_C()` is in the operations after the second `spu_sel`. Note that an SPE executable can issue a 'write' operation to an arbitrary LS address. This is because the SPE has no access permission control (e.g., read-only, executable, or read-write) for the LS. Thus, the inline assembler allows us to use self-modifying code. In the current implementation, `safe_mfc_get_C()` always replaces the quadword starting at `label_1` with `op1` or `op2`, depending on the result of the second `spu_sel` in Line 7. Line 8 writes the code at the `label_1`, but there is a latency for this update of 6 clock cycles. If the DMA operations are safe, then it executes a 'nop' instruction, but if not, then it executes a 'halt' instruction. In comparison with the first approach, both the second and the third approaches have no branch-miss penalties. The second and third approaches are effective when we cannot know the branch probabilities ahead of time or need to eliminate prediction-miss penalties as well as branch penalties.

The three branch optimization techniques introduced

here do not change the functionalities of our original buffer overflow elimination procedures, `safe_mfc_get()` and `safe_mfc_getl()`. Thus, we believe the same buffer overflows will be prevented with those approaches. We will study the performance of those optimization techniques in the next section.

5. Experimental Results

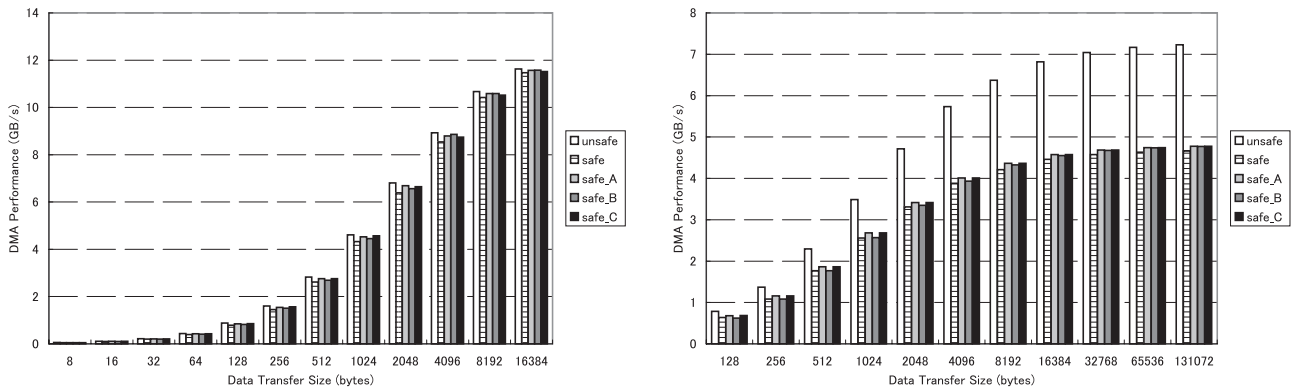
This section presents empirical results for the performance overhead of our safe `mfc_get()` and `mfc_getl()`. In this experiment, we use a micro-DMA benchmark program, `dmabench`, provided as a part of the IBM SDK for Multicore Acceleration. The throughputs of single-block DMA transfers and list-form DMA transfers are evaluated on a 2.8 GHz IBM PowerXCell 8i processor [19] while varying the data transfer sizes from 8 bytes to 16,384 bytes. The graphs in Fig. 5 compare our safe DMA library calls and the unsafe ones. The throughputs in the left-hand graph are averages of 1,000 single-block DMA transfers, and all of the standard deviations were less than 2.03% of each mean. The results in the right-hand graph are averages of 1,000 list DMA transfers, and all of the standard deviations were less than 1.05% of each mean.

The white bars denote the throughputs of the unsafe DMA functions, `mfc_get()` or `mfc_getl()`. The striped bars are the DMA throughputs of `safe_mfc_get()` and `safe_mfc_getl()`, and the basic secure implementation. The light gray, dark gray, and black bars are the performance results of our optimized implementations: branch-hint based optimization, jump-table based optimization, and self-modifying-code based optimization respectively.

In the worst case, we found 10.14% throughput degradation in non-optimized `safe_mfc_get()`. In contrast, we have got at most 3.21% throughput degradation in the branch-hint based optimization, 6.75% throughput degradation in the jump-table based one, and 2.93% throughput degradation in the self-modifying version. The results showed that these three optimization implementation improves the performance overhead compared with non-optimized `safe_mfc_get()`.

We note that the throughput of the self-modifying-code based optimization is better than the other optimizations if the DMA block size is less than 2,048 bytes. With 2,048-byte DMA transfers, branch-hint-based optimization has the best performance. When the DMA transfer size is from 4,096 bytes to 16,384 bytes, jump-table-based optimization is better than the other optimizations. Therefore, if the DMA transfer size is known at the application build time, it is possible to make a compiler selectively insert the branch-hint based optimization code, the jump-table based optimization code, or the self-modifying optimization code based on the DMA transfer sizes. We are studying such improvements for future work.

Table 2 shows the configuration of the list DMA transfers in this experiment. Thanks to our optimizations, we were able to get 6% performance improvement in comparison with the non-optimized `safe_mfc_getl()` when the number of DMA list elements is small. However, the bigger the list element size becomes, the larger the overhead of our safe list-DMA functions is (Fig. 5 (b)). Looking at the safe list-DMA pseudocode in Fig. 3 (b), we compute the total DMA size by extracting the DMA list structure. This computation is the most time-consuming part when we input long DMA lists. For this reason, our optimization methods offer little performance benefit in the long DMA list cases. It might be possible to improve the performance of our safe DMA list transfer code, but the data format conversion from `mfc_list_element` to a vector data type is required to perform SIMD operations, which may cause additional overhead. Improving safe list DMAs is a challenge for the future.



(a) Single-block DMA throughputs (b) List DMA throughputs

Fig. 5 The performance overhead for our secure implementation of DMA functions.

Table 2 The configuration of the list DMA transfers.

Transfer size for each element	128	128	128	128	128	128	128	128	128	128	128
The number of list elements	1	2	4	8	16	32	64	128	256	512	1024
Total transfer size	128	256	512	1024	2048	4096	8192	16384	31768	65536	131072

6. Conclusions and Future Work

We presented a new approach to cope with DMA buffer overflows and overrun attacks. In our approach, unsafe DMA transfer functions are replaced with safe ones at build time, as in Libsafe or libparanoia. In SPE programming, we should address both stack buffer overflows and heap buffer overflows at the same time, which is a special problem of the Cell/B.E. processor. To address these problems, our safe DMA functions evaluate both buffer overflow cases and terminate SPE applications prior to the malicious code injection. Also, we proposed and implemented several optimization techniques for branch reduction or elimination to minimize the performance overhead of buffer overflow elimination. The empirical results showed that we could prevent buffer overflows and overrun attacks with minimal overhead, only a maxium of 2.93% performance degradation.

It would be possible to generate more optimized code by directly modifying the SPE application compilers. If a particular DMA copy never causes an overflow, such as when the DMA copy size is statically defined, we could use the original DMA utility function with no speed penalty. Integrating our DMA buffer overflow elimination into compilers is also future work.

References

[1] 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," Proc. 7th USENIX Security Conference, pp.63–78, San Antonio, Texas, Jan. 1998.

[2] H. Etoh and K. Yoda, "propolice: Improved stack-smashing attack detection," IEICE Technical Report, ISEC2001-43, July 2001.

[3] grsecurity, "Pax." http://pax.grsecurity.net/

[4] A. van de Van, "New security enhancements in red hat enterprise linux v.3, update 3." http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf, 2004.

[5] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation cell processor," Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International, vol.1, pp.184–592, Feb. 2005.

[6] Sony Computer Entertainment, Inc., "PlayStation®3." http://www.us.playstation.com/PS3/Systems

[7] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, S. Daijavaa, and B. Erickson, "Real-time mutual-information-based linear registration on the cell broadband engine processor," Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on, pp.33–36, April 2007.

[8] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," SC '08: Proc. 2008 ACM/IEEE Conference on Supercomputing, pp.1–11, Piscataway, NJ, USA, 2008.

[9] G. Goldrian, T. Huth, B. Krill, J. Lauritsen, H. Schick, I. Ouda, S. Heybrock, D. Hierl, T. Maurer, N. Meyer, A. Schafer, S. Solbrig, T. Streuer, T. Wettig, D. Pleiter, K.H. Sulanke, F. Winter, H. Simma, S. Schifano, and R. Tripiccione, "Qpace: Quantum chromodynamics parallel computing on the cell broadband engine," Computing in Science & Engineering, vol.10, no.6, pp.46–54, Nov.-Dec. 2008.

[10] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," Proc. USENIX Annual Technical Conference, pp.251–262, 2000.

[11] A. Snarskii, "libparanoia." http://www.lexa.ru/snar/libparanoia/

[12] "Cell Broadband Engine Architecture Version 1.02." http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/CBEA_v1.02_11Oct2007_pub.pdf

[13] "GNU Toolchain 4.1.1 and GDB for the Cell BE's PPU/SPU." http://www.bsc.es/projects/deepcomputing/linuxoncell/

[14] "XL C/C++ for Multicore Acceleration, V9.0." http://publib.boulder.ibm.com/infocenter/cellcomp/v9v111/index.jsp

[15] "SPU Application Binary Interface Specification Version 1.5.1." http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/.pdf

[16] T. de Raadt, "Exploit mitigation technique." http://www.openbsd.org/papers/ven05-deraadt/index.html

[17] "Execution disable bit and enterprise security." http://www.intel.com/technology/xdbit/

[18] "Enhanced virus protection." http://www.amd.com/us/products/technologies/enhanced-virus-protection/Pages/enhanced-virus-protection.aspx

[19] IBM Corporation, "PowerXCell 8i Hardware Initialization Guide." http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/53991AEE3346F21E0025751A0015539F/$file/PXC8i_HIG_V1.2_8Dec2008.pdf

Masana Murase received a B.S. in environment and information studies from Keio University and an M.S. in information technologies from the Graduate School of Media and Governance, Keio University. He joined IBM in 2003 in IBM Research - Tokyo. His research interests include system software, system software security, ubiquitous computing, and sensor networks.