Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 66 (2006) 1129–1136
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks

Zili Shao[a,*], Jiannong Cao[a], Keith C.C. Chan[a], Chun Xue[b], Edwin H.-M. Sha[b]

[a]*Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong*
[b]*Department of Computer Science, University of Texas at Dallas, Richardson, Texas 75083, USA*

Received 17 December 2005; received in revised form 31 March 2006; accepted 10 April 2006
Available online 14 June 2006

Abstract

Malicious intrusions by buffer overflow attacks cause serious security problems and pose serious threats for networks and distributed systems such as clusters, Grids and P2P systems. Array & pointer boundary checking is one of the most effective approaches for defending against buffer overflow attacks. However, a big performance overhead may occur after boundary checking is applied. Typically, it may cause 2–5 times slowdown [T.M. Austin, E.B. Scott, S.S. Gurindar, Efficient detection of all pointer and array access errors, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994, pp. 290–301; R.W.M. Jones, P.H.J. Kelly, Backwards-compatible bounds checking for arrays and pointers in c programs, in: The Third International Workshop on Automated and Algorithmic Debugging, 1997, pp. 13–26]. In this paper, we propose a hardware/software method to optimize the performance of array & pointer boundary checking by designing a special boundary checking instruction. The experimental results show that our method can effectively reduce the overhead of array & pointer boundary checking.

Keywords: Buffer overflow; Hardware/software optimization; Array & pointer boundary checking

1. Introduction

Buffer overflow attacks are one of the commonest methods for attackers to break into systems. By exploiting buffer overflow vulnerabilities, attackers can control computer systems in networks or in distributed systems such as Grids, clusters, P2P systems, and launch various attacks such as DDoS (distributed denial-of-service). A typical example is MS Blaster, a notorious worm that occurred in 2003. The MS Blaster worm exploits a buffer overflow vulnerability in Microsoft DCOM RPC and infected millions of computer systems in the world. A DDoS attack was launched by the MS Blaster worm against Microsoft's Windows Update website [15]. Another notorious worm in 2003, *Sapphire* (or *SQL Slammer*), also took advantage of buffer overflow vulnerabilities to break into systems [2]. Malicious intrusions by buffer overflow attacks may destroy valuable hosts, clusters, Grids, P2P systems, etc. and pose serious threats for networks and distributed systems. Therefore, how to defend against buffer overflow attacks becomes an important problem in Internet-based environments.

Various approaches have been proposed to protect systems against buffer overflow attacks. To defend against stack overflow attacks, techniques such as StackShield, StackGuard, IBM SSP, StackGhost, etc., have been proposed to guard stacks at runtime by adding extra instructions assisted by compilers [4]. To defend against pointer-corruption attacks, the techniques such as hardware/software co-design protection [14,13], PointGuard [3], etc., are proposed by encrypting pointer values while they are in memory and decrypting them before dereferencing. The static checking method detects the vulnerabilities by analyzing the C source code using software tools [16]. The dynamic checking method uses program testing strategy that checks buffer overflow vulnerabilities by executing programs with specific inputs [5]. All the above techniques cannot completely protect systems against buffer overflow attacks.

* Corresponding author.
E-mail address: cszlshao@comp.polyu.edu.hk (Z. Shao).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.04.010

So far, array & pointer boundary checking is one of the most effective protection approaches against buffer overflow attacks. In array & pointer boundary checking, instructions are added to check the bounds of arrays and pointers at runtime [9,1,8]. Since every array & pointer dereference is ensured to be in bounds, overflow cannot be caused. While this strategy may completely protect systems against buffer overflow attacks, a big performance overhead may occur especially for array and pointer intensive applications. Typically, it may cause 2–5 times slowdown after boundary checking is applied [1,8].

To reduce the performance overhead of array & pointer boundary checking, two different optimization approaches have been proposed through software and hardware, respectively. The software optimization method reduces the execution time mainly through elimination of redundant checks and propagation of checks out of loops [6]. Although the techniques based on this approach can greatly reduce the number of boundary checks, the overhead for some applications is still big. For example, in [6], after the optimization is applied, there are still 10–60% of checks left.

The hardware optimization method reduces the execution time by parallel performing checking with additional hardware. In [11], a sophisticated technique is proposed to optimize the performance by using a second CPU to perform checking in parallel with very low overhead. However, the proposed technique is complicated and expensive in terms of hardware cost. In [14,13], we propose a hardware/software codesign method with special secure instructions to defend against buffer overflow attacks. However, the secure instructions are designed based on the idea to encrypt and decrypt return addresses and pointers, and cannot be used to improve the performance of array & pointer boundary checking.

The software and hardware optimization methods are separated in the previous work. Therefore, the good results cannot be achieved. The main reason that the software optimization method cannot further reduce the overhead is that it lacks necessary hardware support. As we will show in Section 2, it needs two comparison instructions and two branch instructions to finish one check. And it takes at least six clock cycles in the DLX architecture, a generic RISC CPU with five-phase pipeline [12]. Given such big overhead for each check, it is hard for software optimization techniques to achieve good performance. On the other hand, without taking any advantage of software optimization, it is not surprising that the techniques based on the pure hardware optimization is complicated and with too much hardware cost. In this paper, we attempt to bridge the gap between the software optimization and hardware optimization, and improve the performance of array & pointer boundary checking with the focus on protecting systems against buffer overflow attacks.

This paper proposes a hardware/software method to optimize the performance of array & pointer boundary checking. We first design a special instruction to efficiently perform boundary checking. We then propose the optimization strategy combining with the existing software optimization approaches, and discuss our C-to-C transformation strategy for implementing array & pointer boundary checking in the C language in terms of defending against buffer overflow attacks. Finally, we experiment with the hardware/software optimization approach on the SimpleScalar/ARM Simulator [10]. The experimental results show that the overhead of array & pointer boundary checking can be greatly reduced. Our conclusion is that with careful hardware/software optimization, array & pointer boundary checking can be a practical solution for defending systems against buffer overflow attacks with tolerable overhead.

The rest of this paper is organized as follows. Section 2 gives several motivational examples and provides the necessary background. The special boundary checking instruction and the performance analysis are presented in Section 3. The optimization strategy combining with the existing software optimization approaches is discussed in Section 4. The implementation and experiments are presented in Sections 5 and 6, respectively. Section 7 concludes this paper.

2. Background

In this section, we show examples to provide the background for understanding why we need to use array & pointer boundary checking to defend systems against buffer overflow attacks. We first show an example in which an attack is initiated through a string variable. We then show an example with boundary checking protection. In these examples, we use Intel $\times 86$ processors as the platform because it is easy to explain and understand.

Our first example shows that it is not enough to protect return addresses and pointers. As shown in Fig. 1(a), Vulnerable program uses *strcpy*() to copy inputs to buffer[], and then the content in buffer[] is copied into a temporary file "/tmp/001/001.tmp". An attack can be deployed as shown in Fig. 1(b), where the string variable containing the file is overflowed and the content is changed as "/root/.rhosts" when "strcpy(buffer,msg)" is executed. Then "/root/.rhosts" will be opened when "tmpfd=fopen(tmpfile, "w")" is executed. It follows that the inputs will be written into "/root/.rhosts" instead of "/tmp/001/001.tmp". With the crafted inputs, then an attacker will gain the control of a computer with "root" privilege. This kind of attacks neither overwrites return addresses nor changes pointers. So it cannot be defended by combining stack protection and pointer protection techniques.

As shown in the above example, to protect return address, pointers, etc., cannot completely defend systems against buffer overflow attacks. Array & pointer boundary checking solves the problem by adding instructions to check array bounds and perform pointer checking at runtime. To ensure every array & pointer dereference to be in bounds, the overflow cannot be caused. Various strategies have been proposed to add boundary checking [9,1,8]. To defend against buffer overflow attacks, the biggest concern is performance overhead. In this paper, we use a simple approach that carries boundary information with each pointer similar to [9]. Basically, a triple-address structure called enhanced pointer is associated with each pointer, in which the information about the base address, the lower-bound address, and upper-bound address of a pointer is included.

An example to add boundary checking by our transformation strategy (discussed in Section 5) is shown in Fig. 2. Fig. 2(a)

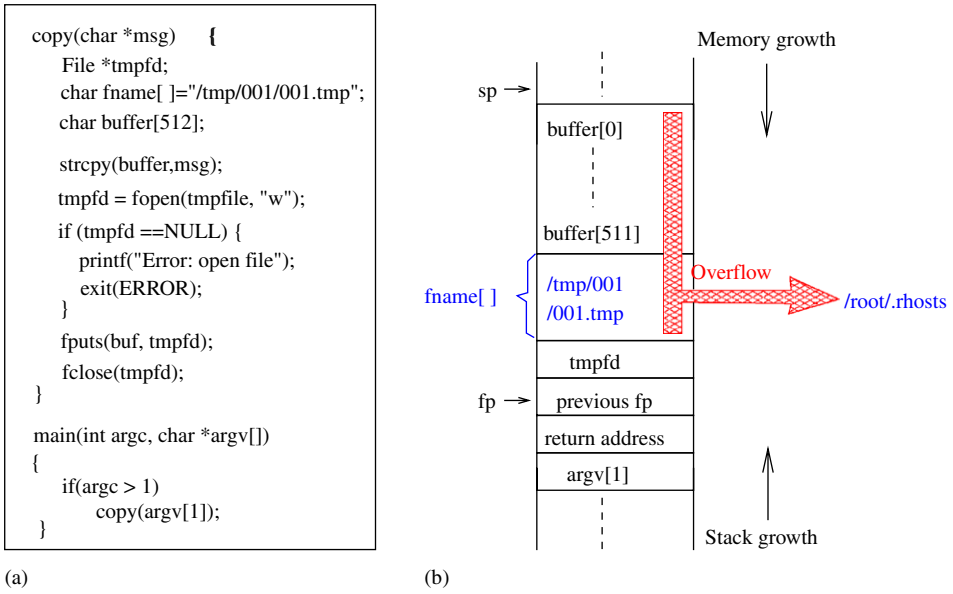

Fig. 1. (a) Vulnerable program. (b) The stack structure and an attack through a string variable.

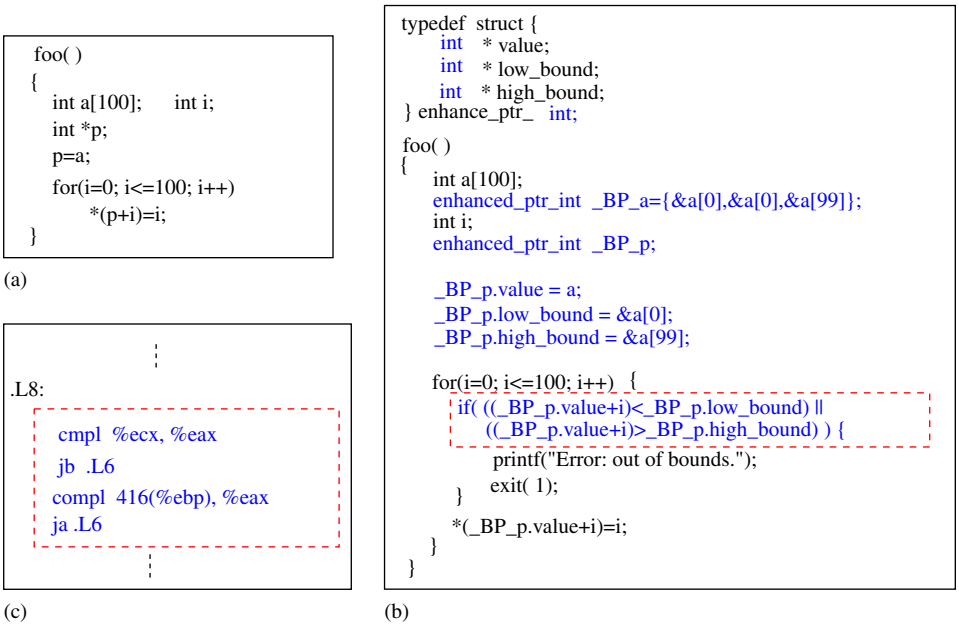

Fig. 2. (a) An example C program. (b) The program with boundary checking. (c) The corresponding assemble code to do boundary checking compiled by GCC with complete optimization.

shows an example C program and Fig. 2(b) shows the program with boundary checking. The enhanced pointers with prefix flag "_BP_" in Fig. 2(b) are obtained from the integer pointer (array) in Fig. 2(a). The enhanced pointer for the array is declared and initialized to contain its bound data. The enhanced pointer for the pointer is declared and replaces the original pointer at all places. The boundary checking statements are inserted before the dereference of a pointer that is related to "write" operation. If the dereference is out of bounds, it will be detected and the program will halt and exit. For this program, the last operation, "*(_BP_p.value+100)=i", is out of bounds. And this out-bound violation will be caught in the program with boundary checking.

To consider the execution time of each check, Fig. 2(c) shows the corresponding assembly instructions to finish one check that is compiled by GCC with full optimization on Intel ×86 platform. Totally, it needs four instructions: two comparison instructions and two branch instructions, in which the object address is stored in register "%eax", and the lower bound and upper bound are stored in register "%ecx" and in the address pointed by "−416(%ebp)". Obviously, it is too much to finish one check using four instructions.

From the above example, we can see there are mainly two source of overhead caused by adding array & pointer boundary checking: the instructions to set the lower-bound and upper-bound addresses, and the instructions to do boundary checking. The number of the boundary-setting instructions is closely related to the number of checks especially for array and pointer intensive applications. Therefore, there are mainly two ways to reduce the total overhead from boundary checking:

(1) Reduce the total number of checks.
(2) Reduce the execution time to finish one check.

While the reduction of total number of checks has been studied from the previous work, a little work has been done for the latter. Therefore, in this paper, we study these problems. In the next section, we will discuss how to reduce the execution time of one check. We then discuss how to combine our method with the previous optimization methods in Section 4.

3. Special bounds checking instruction

In this section, we propose a new instruction to perform boundary checking in order to reduce the execution time of one check. We first analyze the execution time of performing one boundary checking on DLX architecture. Then we design a new instruction called BCK to perform boundary checking. The semantics and the implementation of the new instruction are introduced. In order to make our analysis and design be general and easily extended to various platforms, our analysis and design are based on the DLX architecture, a generic RISC CPU with typical pipeline structure.

The DLX architecture [12] has a five-phase pipeline: IF, ID, EX, MEM, and WB. On the DLX architecture, it needs two comparison and two branch instructions to finish one boundary check similar to the assembly code shown in Fig. 2(c). Assume that the object address, the lower-bound address and the upper-bound address are in register R1, R2, R3, respectively. Fig. 3 shows the execution to perform one boundary check on DLX architecture, in which instruction "*SLT R1, R2, R4*" means "$R4 \leftarrow 1$ if $R1 < R2$; otherwise $R4 \leftarrow 0$", and instruction "*BNER R4,Out_Bounds*" means "*jump to Out_Bounds if R4* $\neq 0$". In the pipeline architecture, branch operations are assumed to be processed in the early phase (ID), so it takes one pipeline stall for each branch operation. Even with such branch optimization, we can see that it takes six clock cycles to finish one check with the four instructions. Given such big overhead for one check, it is hard for software optimization techniques to achieve good performance especially in array & pointer intense applications.

To reduce the overhead for one check, we need a special instruction that can efficiently perform boundary checking. In the DLX architecture (and RISC CPUs), there is no operand that can contain memory address in typical ALU instructions. Thus, this special instruction needs to take three registers that store three addresses (the object address, the lower-bound address, the upper-bound address) as the inputs. To avoid pipeline stall, there should have no control hazard between the new boundary checking instruction and the next instruction that has deference operation for a object address. It can be solved with a special exception as the output, since array & pointer boundary checking can be dealt with the same way: if it is in bounds, do nothing; otherwise, an out-bound exception handling program is called to do recovery or exit. Therefore, our special boundary checking instruction called BCK in the DLX architecture has the format and meaning as shown in Table 1.

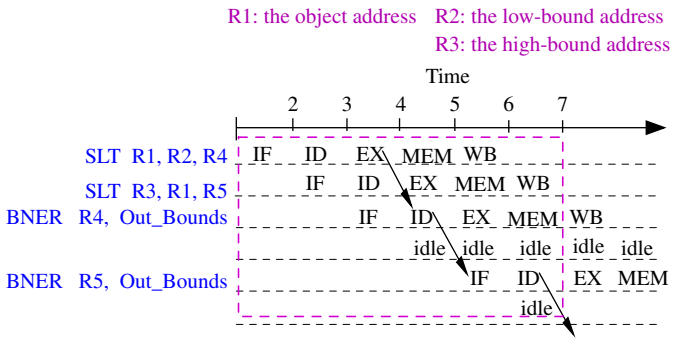

Fig. 3. Six clock cycles needed to perform one boundary check in the DLX architecture without using the special boundary checking instruction.

Table 1
The new boundary checking instruction *BCK* in DLX

Example instruction	Meaning
BCK R1,R2,R3	if ((R1 < R2) || (R1 > R3)) Generate *Out_Bounds* exception signal

Using the exception as the output in *BCK*, the following instruction can be fetched and executed without pipeline stall. If *BCK* causes an exception by an out-bound object address, then the pipeline is flushed at the end of EX phase; therefore, the following instruction cannot cause any memory/register change, as it is at the end of ID phase at that time. As shown in Fig. 4, to implement instruction *BCK* in DLX, the ID/EX register needs to be expanded so it can contain one extra operand. And one extra ALU is needed to perform the extra comparison in parallel. For commercial RISC CPUs, the extra hardware might not be needed, since their MMUs (memory management unit) are powerful to finish such simple comparison operations if they support virtual memory.

If this bound checking instruction is applied in optimization, a big performance improvement can be achieved. The extra instructions of setting bound information for a check can usually be hoist out of loops with software optimization. So in the DLX architecture, it achieves about 83.3% reduction in the execution time and about 75% reduction in the number of instructions for one check.

4. The optimization

In this section, we discuss the optimization strategy combining with the existing software optimization approaches. We first introduce how the software optimization reduces the overhead of pointer & array checking. Then we discuss how to fully

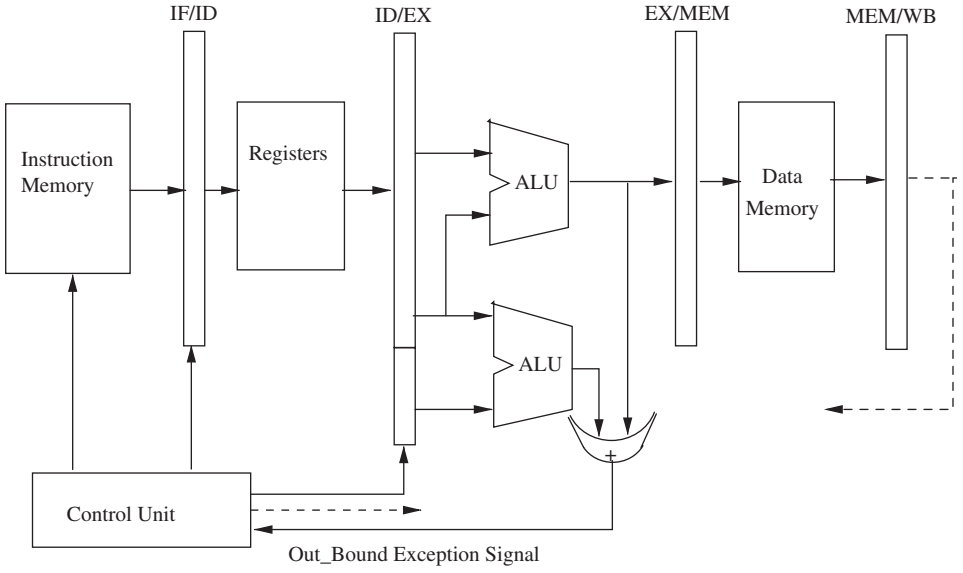

Fig. 4. The implementation of the special bound checking instruction in DLX.

take advantage of the special boundary checking instruction to optimize the performance.

The previous software optimization mainly focuses on optimizing *array* boundary checking. The basic idea is to reduce the number of checks by eliminating redundant checks and hoisting checks out of loops [6]. To change check locations, the point reported from an error and the point at which the array boundary violation really occurs may not be the same. This may not be good for catching memory errors. However, this is fine for defending against buffer overflow attacks, where the security and performance are the biggest concerns.

The extension to optimize pointer boundary checking by this method is straightforward. The similar analyzing method can be applied to eliminate checks and hoist checks out of loops for pointer checks. For example, for the program with boundary checking shown in Fig. 2(b), we can move the checks out of loops by only considering the minimum and maximum values of *i* that associates with the deference of the pointer. In this way, we only need to do two checks as shown in Fig. 5 compared with 101 checks done in the program in Fig. 2(b). However, the calculation of the scopes related to a pointer or array may not be always so easy. For some applications, the number of checks left after the kind of optimization may be still big. Therefore, the further optimization by utilizing the special boundary check instruction is needed.

The new boundary checking instruction can be easily utilized to optimize the performance by a compiler. We only need to put a special prefix flag to distinguish boundary checking sentences and ordinary comparison when inserting checks into a program. For example, in Fig. 5, the flag "_BP_" is associated with the variables in boundary checking sentences. In fact, this has been applied in almost every array & pointer boundary checking approaches (and is an easy revision if they do not have). In this way, a compiler can identify boundary checking sentences and translate them by using the new boundary checking instruction.

```
foo( )
{
   int a[100];
   enhanced_ptr_int  _BP_a={&a[0],&a[0],&a[99]
   int i;
   enhanced_ptr_int  _BP_p;

   _BP_p.value = a;
   _BP_p.low_bound = &a[0];
   _BP_p.high_bound = &a[99];

  if( ( (_BP_p.value+0) < _BP_p.low_bound ||
      (_BP_p.value+0) > _BP_p.high_bound  ) ||
    ( (_BP_p.value+100) < _BP_p.low_bound ||
     (_BP_p.value+100) > _BP_p.high_bound )  ){
  {
       printf("Error: out of bounds.");
       exit( 1);
  }

  for(i=0; i<=100; i++)
       *(_BP_p.value+i)=i;
}
```

Fig. 5. The optimization by putting boundary checking out of loops.

Combining our special boundary checking instruction with software optimization, an overall performance improvement can be achieved by the reduction of the number of checks (the C source-code-level software optimization) and the reduction of the execution time of one check (the special boundary checking instruction).

5. Implementation

The optimization methods discussed in Section 4 are general and can be applied to reduce the overhead for various array & pointer boundary checking approaches. Almost all boundary checking approaches in the previous work focus on checking memory access errors. Although they can effectively

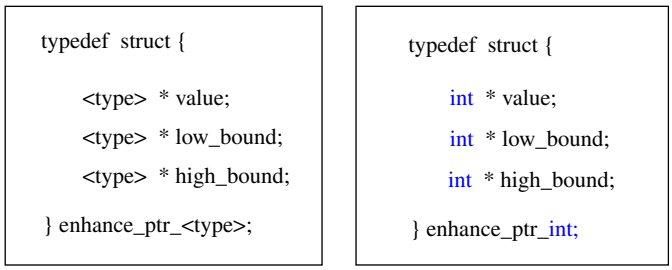

Fig. 6. (a) The general structure of an enhanced pointer. (b) The structure of an enhanced integer pointer.

defend against buffer overflow attacks, they are not very efficient at runtime. So in this section, we discuss our C-to-C transformation strategy for implementing boundary checking against buffer overflow attacks. We first propose our checking policy, and then discuss the representation and transformation of arrays and pointers considering the inter-operation with unprotected code.

Regardless which type of buffer overflow attacks, buffers have to be overflowed for an attack to succeed. If all write operations associated with pointers and arrays are ensured to be in bounds, then overflow cannot occur. Therefore, we only need to check the bounds of a pointer when its dereference is related to "write" operations for defending against buffer overflow attacks. In our checking policy, checks are only inserted before the dereference of a pointer that is related to "write" operation.

To serve this purpose, we can use a simple method to carry boundary information with each pointer similar to the method used in BCC, RTCC, bcc, etc. A prototype of our enhanced pointer is shown in Fig. 6(a), which is a structure containing three addresses: a pointer itself, the lower and upper addresses of a pointer. In the structure, "⟨type⟩" will be replaced with the specific types associated with pointers in a program. For example, Fig. 6(b) shows a structure of an enhanced integer pointer. Our transformation policies for pointers and arrays are shown as follows.

5.1. The transformation for pointers

Declaration: Given a pointer in an original program, an enhanced pointer is declared with the pointer as well as its two boundary addresses as shown in Fig. 2(b) in Section 2.

Replacement: An enhanced pointer replaces the original pointer at all places in the transformed program with its "value" item in the structure.

Assignment: Each assignment of a pointer in an original program is transformed as an initiation of the corresponding enhanced pointer in the transformed program. For example, in Fig. 6(b), the enhanced pointer is initiated corresponding to the assignment of "p=a" in the original program (Fig. 2(a)), where its pointer value (_BP_p_int.value) is pointed to "a[0]", its lower bound and upper bound (_BP_p_int.low_bound and _BP_p_int.high_bound) are set as the bounds of array "a[]". The boundary data of the enhanced pointer will be changed along with each assignment so it can keep the same boundary information as the assigned array or pointer.

Parameter passing with inter-operation: For parameter passing, there will be no inter-operation problem, if the "value" item of an enhanced pointer is used to directly replace the pointer in a function call. For example, in an original program, if there is a function, "Function(p,...)", with pointer "p" as the parameter, then "Function(_BP_p.value,...)" works well without any change after we replace "p" with the "value" item of the enhanced pointer, "_BP_p.value" ("p" and "_BP_p.value" has the same type). In this way, we can deal with unprotected code.

Parameter passing with bound information: However, to serve our purpose, we should pass an enhanced pointer instead of the pointer in a function call. So the boundary checking can be performed in the function with the boundary information associated with an enhanced pointer. In this way, the interface of functions needs to be changed.

5.2. The transformation for arrays

Declaration: Given an array in an original program, an enhanced pointer is declared, and inserted after the declaration of the array in the transformed program (see Fig. 2 in Section 2 as an example).

Initialization: An enhanced pointer for an array is initiated when it is declared. For example, the enhanced pointer for the array is initialized as shown in Fig. 2(b). In the example, its pointer value (_BP_a_int.value) is pointed to the first element of the array (&a[0]), its lower bound and upper bound (_BP_a_int.low_bound and _BP_a_int.high_bound) are pointed to the bounds of the array (&a[0] and &a[99]), respectively. After that, the boundary information of the enhanced pointer will not be changed anymore.

Replacement: All arrays are kept without change in the transformed program except the case discussed in the next item.

Parameter passing: Similar to the transformation for pointers, if we want the inter-operation, we can keep arrays as parameters without any change; otherwise, we use an enhanced pointer to pass the boundary data. For example, assume "a[]" is an array, "_BP_a" is the corresponding enhanced pointer, and a function, "copy1(&a[10])", is called in the original program. Then we assign the address of "a[10]" to the "value" item of the enhanced pointer (_BP_a.value=&a[10]) and pass the enhanced pointer as "copy1(_BP_a)" in the transformed program.

In summary, to defend against buffer overflow attacks, we need to pass the boundary information of arrays & pointer if there are such parameters in functions, since it is possible that an attack will be launched by overflowing an array pointed by passed parameter. For example, the vulnerable library function, strcpy(), can be exploited to overflow the array passed to it. On the other hand, based on our policy, it is very easy to interoperate with unprotected code such as legacy software. With very little overhead, our method can be applied to protect the whole system including the kernel.

6. Experiments

In this section, we experiment with our array & pointer boundary checking method on a small set of programs. In order to obtain cycle-accurate measurement, we use the SimpleScalar/ARM Simulator [10] configured as the StrongARM-110 microprocessor architecture as our test platform. The simulator is installed on a Dell computer with Intel Pentium 4 CPU running Red Hat Linux 9.

The benchmarks are shown in the first column in Table 2. The first benchmark is from the example shown in Fig. 2. FFT (fast fourier transform), IFFT (inversed FFT) and String Search are selected from MiBench, a free, commercially representative embedded benchmark suite [7]. MatMpy is a benchmark to compute the multiplication of two integer matrices with 20×20 (small) and 200×200 (large). QuickSort is quick sort program with the inputs of 5000 integers (small) and 50,000 integers (large). BubbleSort is bubble sort program with the inputs of 1000 integers (small) and 5000 integers (large).

For each benchmark, we compare the time performance of the original program, the protected program without optimization, and the protected program with optimization. When adding boundary checking statements, we apply our checking policy that only checks the deference associated with "write" operations. Boundary checking statements are added using the methods similar to bcc [9]. A protected program without optimization is a program in which our array & pointer boundary checking approach in Section 5 is directly applied. A protected program with optimization is the program using the software optimization method. The experimental results are shown in Table 2.

Table 2
The comparison of the execution time of the original programs and the protected programs

Benchmarks	Original	Protected without Opti.		Protected with Opti.	
	Time (cycles)	Time (cycles)	Overhead (%)	Time (cycles)	Overhead (%)
The example (Fig. 2)	37,349	39,022	4.48	37,711	0.97
FFT (small)	78,696,891	81,596,639	3.68	79,164,981	0.59
FFT (large)	471,268,795	912,067,050	93.53	477,295,365	1.28
IFFT (small)	137,057,269	140,622,324	2.60	140,622,282	2.60
IFFT (large)	472,513,514	716,907,862	51.72	478,710,497	1.31
String Search	5,402,119	9,752,968	80.54	5,970,558	10.52
MatMpy (small)	1,495,814	1,596,768	6.75	1,517,411	1.44
MatMpy (large)	278,454,909	389,366,176	39.83	315,239,739	13.21
QuickSort (small)	18,715,285	20,016,684	6.95	19,867,346	6.16
QuickSort (large)	196,385,271	212,044,087	7.97	210,545,337	7.21
BubbleSort (small)	29,191,945	61,473,455	110.58	31,473,172	7.81
BubbleSort (large)	695,390,264	1,502,284,272	116.03	752,776,830	8.25
Average overhead over original prog.			43.72		5.11

From Table 2, we can see that the overhead is small. For programs protected by boundary checking without optimization, the average overhead is 43.72%. Compared with the previous work that has 2–5 times overhead, the improvement comes from our write-only checking policy. The average overhead is 5.11% with the optimization. In the experiments, we did not apply special boundary checking instruction in the optimization. The further performance improvement can be achieved in processors supporting this special boundary checking instruction. With such low overhead, array & pointer boundary check can be a practical solution to defend against buffer overflow attacks.

7. Conclusion

Array & pointer boundary checking is one of the most effective protection approaches for defending against buffer overflow attacks. However, a big performance overhead may occur after boundary checking is applied from the previous work. In this paper, we proposed a hardware/software method to optimize the performance of array & pointer boundary checking. We first proposed a special instruction to efficiently perform boundary checking. We then propose the optimization strategy combining with the existing software optimization approaches. Finally, we discussed our C-to-C transformation strategy for implementing array & pointer boundary checking in terms of defending against buffer overflow attacks, and conducted experiments with the hardware/software optimization. The experimental results show that the overhead of array & pointer boundary checking can be greatly reduced. Our conclusion is that with careful hardware/software optimization, array & pointer boundary checking can be a practical solution for defending systems against buffer overflow attacks with tolerable overhead. In this paper, the techniques we discussed are based on C/C++ program languages. How to apply them in interpretive languages such as Java will be one of our future research topics.

References

[1] T.M. Austin, E.B. Scott, S.S. Gurindar, Efficient detection of all pointer and array access errors, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994, pp. 290–301.

[2] CERT/CC, CERT Advisory CA-2003-04 MS-SQL Server Worm, World Wide Web, 2003. URL ⟨http://www.cert.org/advisories/ CA-2003-04.html⟩.

[3] C. Cowan, S. Beattie, J. Johansen, P. Wagle, Pointguard: protecting pointers from buffer-overflow vulnerabilities, in: Proceedings of the USENIX Security Symposium, 2003.

[4] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grie, P. Wagle, Q. Zhang, Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks, in: Proceedings of the USENIX Security Symposium, 1998.

[5] G. Fink, C. Ko, M. Archer, K. Levitt, Towards a property-based testing environment with applications to security-critical software, in: Proceedings of the Fourth Irvine Software Symposium, 1994.
[6] R. Gupta, Optimizing array bound checks using flow analysis, ACM Lett. Programming Languages Syst. 2 (1–4) (1993) 135–150.
[7] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, Mibench: a free, commercially representative embedded benchmark suite, in: IEEE Fourth Annual Workshop on Workload Characterization, Austin, TX, 2001.
[8] R.W.M. Jones, P.H.J. Kelly, Backwards-compatible bounds checking for arrays and pointers in c programs, in: The Third International Workshop on Automated and Algorithmic Debugging, 1997, pp. 13–26.
[9] S.C. Kendall, Bcc: Runtime checking for c programs, in: Proceedings of the Summer USENIX Conference, 1983.
[10] S. LLC, SimpleScalar/ARM, World Wide Web, 2000. URL ⟨http://www.eecs.umich.edu/taustin/code/arm/simplesim-arm-0.2.tar.gz⟩.
[11] H. Patil, C. Fischer, Low-cost, concurrent checking of pointer and array accesses in c programs, Software practice and experience.
[12] D.A. Patterson, J.L. Hennessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publisher, Los Altos, CA, 1996.
[13] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, E.H.-M. Sha, Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software, IEEE Trans. Comput. 55 (4) (2006) 443–453.
[14] Z. Shao, Q. Zhuge, Y. He, E. H.-M. Sha, Defending embedded systems against buffer overflow via hardware/software, in: IEEE 19th Annual Computer Security Applications Conference, Las Vegas, 2003, pp. 352–361.
[15] Symantec Security Response, W32.Blaster.Worm, World Wide Web, 2005. URL ⟨http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm. html⟩.
[16] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, in: Network and Distributed System Security Symposium, San Diego, CA, 2000, pp. 3–17.

Zili Shao received his B.E. Degree in Electronic Mechanics from the University of Electronic Science and Technology of China, China, 1995. He received his M.S. and Ph.D. Degrees from the Department of Computer Science at the University of Texas at Dallas, in 2003 and 2005, respectively. He has been an Assistant Professor in the Department of Computing at the Hong Kong Polytechnic University since 2005. His research interests include embedded systems, high-level synthesis, compiler optimization, hardware/software co-design and computer security.

Jiannong Cao received his B.Sc. degree in Computer Science from the Nanjing University, Nanjing, China in 1982, and his M.Sc. and Ph.D. degrees in Computer Science from the Washington State University, Pullman, WA, USA, in 1986 and 1990, respectively.
He is currently a Professor in the Department of Computing at the Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the Director of the Internet and Mobile Computing Lab in the Department. Before joining the Hong Kong Polytechnic University, he was on the faculty of computer science at the James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile and wireless computing, fault tolerance, and distributed software architecture. He has published over 180 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing test-bed, protocols, middleware and applications.
Dr. Cao is a Senior Member of China Computer Federation, a Senior Member of the IEEE Computer Society, the IEEE Communication Society, IEEE, and ACM. He is also a Member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/programme committee member for many international conferences.

Keith C. C. Chan received his B.Math. degree in Computer Science and Statistics, and his M.A.Sc. and Ph.D. degrees in Systems Design Engineering from the University of Waterloo, Waterloo, Ontario, Canada. He had worked for a number of years at the IBM Canada Laboratory, Toronto Ontario, where he was involved in the development of software engineering tools. In 1993, he joined the Department of Electrical and Computer Engineering, Ryerson University, Toronto, Ontario, Canada, as an Associate Professor and in 1994, he joined the Department of Computing of The Hong Kong Polytechnic University, Hong Kong where he is now Professor and Head. He is also a Guest Professor of the Graduate School of The Chinese Academy of Sciences, Beijing, China. He is active in consultancy and has served as a consultant to government agencies and various companies in Hong Kong, China, Singapore, Malaysia and Canada. His research interests are in pervasive computing, data mining, bioinformatics and software engineering.

Chun Xue received his B.S. Degree in Computer Science and Engineering from the University of Texas at Arlington in May 1997, and M.S. Degree in Computer Science from the University of Texas at Dallas, in December 2002. He is currently a Ph.D. candidate in the Department of Computer Science at the University of Texas at Dallas. His research interests include performance and memory optimization for embedded systems, and software/hardware co-design for parallel systems.

Edwin Hsing-Mean Sha received his M.A. and Ph.D. degrees from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, Notre Dame, IN. He served as an Associate Chairman from 1995 to 2000. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas.
He has published more than 210 research papers in refereed conferences and journals. He has served as an Editor for many journals, and as Program Committee Members and Chairs in numerous international conferences. He received Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, and NSF CAREER Award.