Efficient Array & Pointer Bound Checking Against Buffer Overflow Attacks via Hardware/Software *

Zili Shao, Chun Xue, Qingfeng Zhuge, Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA
{zxs015000, cxx016000, qfzhuge, edsha}@utdallas.edu

Bin Xiao
Department of Computing
Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
csbxiao@comp.polyu.edu.hk

Abstract

Buffer overflow attacks cause serious security problems. Array & pointer bound checking is one of the most effective approaches for defending against buffer overflow attacks when source code is available. However, original array & pointer bound checking causes too much overhead since it is designed to catch memory errors and it puts too many checks. In this paper, we propose an efficient array & pointer bound checking strategy to defend against buffer overflow attacks. In our strategy, only the bounds of write operations are checked. We discuss the optimization strategy via hardware/software and conduct experiments. The experimental results show that our strategy can greatly reduce the overhead of array & pointer bound checking. Our conclusion is that based on our strategy, array & pointer bound checking can be a practical solution for defending systems against buffer overflow attacks with tolerable overhead.

1 Introduction

Buffer overflow attacks cause serious security problems. In 2003, 23 out of 28 serious vulnerability reports in CERT/CC Advisories were buffer overflow related: nineteen were directly related to stack overflow or heap overflow, two were related to integer overflow that can cause heap overflow, and two were related to memory deallocation bugs that can also cause heap overflow. The two most notorious worms that occurred in 2003, *Sapphire* (or *SQL Slammer*) and *MSBlaster*, also took advantage of buffer overflow vulnerabilities to break into systems.

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 [3]. To defend against pointer-corruption attacks, the techniques such as hardware/software co-design protection [11], PointGuard [2], 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 [12]. The dynamic checking method uses program testing strategy that checks buffer overflow vulnerabilities by executing programs with specific inputs [4]. All the above techniques can not completely protect systems against buffer overflow attacks.

So far, array & pointer bounds checking is one of the most effective protection approaches against buffer overflow attacks when the source code is available. In array & pointer bounds checking, instructions are added to check the bounds of arrays and pointers at runtime [7, 1]. Since every array & pointer dereference is ensured to be in bounds, overflow can not be caused. So this strategy may completely protect systems against buffer overflow attacks. The proposed array & pointer checking from the previous work mainly focuses on checking memory errors, therefore, checks are put for each memory access including read/write. Therefore, a big performance overhead may occur especially for array and pointer intensive applications. Typically, it may cause 2-5 times slowdown after bounds checking is applied [1].

To reduce the performance overhead of array & pointer bounds 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 [5]. Although the techniques based on this approach can greatly

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, USA, and HK POLYU A-PF86 and COMP 4-Z077, HK.

reduce the number of bounds checks, the overhead for some applications is still big. For example, in [5], 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 [9], 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 expensive in terms of hardware cost.

In this paper, we propose an efficient array & pointer bound checking strategy to defend against buffer overflow attacks. In our strategy, only the bounds of write operations are checked. We discuss our C-to-C transformation for implementing array & pointer bounds checking in the C language based on this strategy. The representation of extended pointers and the transformation of pointers and arrays considering the inter-operation with unprotected code are discussed.

We then propose the optimization strategy via software/hardware. Combining with the existing software optimization approaches, we discuss the strategy to insert bounds checking so a compiler can easily use the new instruction in translation, and propose to reduce the number of instructions that set the low-bound and high-bound addresses as much as possible in code generation. A special bound checking instruction is proposed to efficiently perform bound check and its performance is analyzed under the DLX architecture. Finally, we experiment with our approach. The experimental results show that the overhead of array & pointer bounds checking can be greatly reduced. Our conclusion is that based on our strategy, array & pointer bounds 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. The proposed array & pointer checking approach is presented in Section 2. The software/hardware optimization strategies are discussed in Section 3 and Section 4, respectively. The experiments are presented in Section 5. Section 6 concludes this paper.

2 Efficient Checking and Transformation

Using array & pointer bound checking, every array & pointer dereference is ensured to be in bounds, therefore, overflow can not be caused. So this strategy may completely protect systems against buffer overflow attacks. However, Almost all bound checking approaches in the previous work focus on checking memory access errors. Although they can effectively defend against buffer overflow attacks, they are not very efficient at runtime. In this section, we discuss our C-to-C transformation strategy for implementing bound 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 can not 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.

```
typedef struct EPT_<type> {
    <type> * value;
    <type> * low_bound;
    <type> * high_bound;
} enhance_ptr_<type>;
```

(a)

```
typedef struct EPT_INT{
    int * value;
    int * low_bound;
    int * high_bound;
} enhance_ptr_int;
```

(b)

Figure 1. (a) An enhanced pointer. (b) An enhanced integer pointer.

We use a simple method to carry bound information with each pointer similar to the method used in BCC,RTCC,bcc,etc. A prototype of our enhanced pointer is shown in Figure 1(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, Figure 1(b) shows a structure of an enhanced integer pointer. Our transformation policies for pointers and arrays are shown as follows.

2.1 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 bound addresses as shown in Figure 2(b).
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 Figure 1(b), the enhanced pointer is initiated corresponding to the assignment of "p=a" in the original program (Figure 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 bound data of the enhanced pointer will be changed along with each assignment so it can keep the same bound 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 bound checking can be performed in the function with the bound information associated with an enhanced pointer. In this way, the interface of functions needs to be changed.

2.2 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 Figure 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 Figure 2(b), where 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 bound 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 bound data. For example, assume "a[]" is an array, "_BP_a" is the corresponding enhanced pointer, and a bounded 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.

An example to add bound checking by our transformation strategy is shown in Figure 2. Figure 2(a) shows an example C program and Figure 2(b) shows the program with bound checking. The enhanced pointers with prefix flag "_BP_" in Figure 2(b) are obtained from the the integer pointer (array) in Figure 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 bound

```
foo( )
{
      int a[100];int i;  int *p;

      p=a;
     for(i=0; i<=100; i++)
        *(p+i)=i;
}
```

(a)

```
typedef  struct EPT_INT {
     int   * value;
     int   * low_bound;
     int   * high_bound;
} enhance_ptr_int;
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];

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

(b)

Figure 2. (a) An example C program. (b) The program with bound checking.

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 outbound violation will be caught in the program with bound checking.

In summary, to defend against buffer overflow attacks, we need to pass the bound 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 inter-operate with unprotected code such as legacy software.

3 The Software 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 take advantage of the special bounds checking instruction to optimize the performance. Finally, we propose to reduce the numbers of the instructions that set the lower-bound and upper-bound addresses as much as possible in code generation.

The previous software optimization mainly focuses on optimizing *array* bounds checking. The basic idea is to reduce the number of checks by eliminating redundant checks and hoisting checks out of loops [5]. To change check locations, the point reported from an error and the point at which the array bounds 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.

```
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;
}
```

Figure 3. The optimization by putting bounds checking out of loops.

The extension to optimize pointer bounds 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 bounds checking shown in Figure 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 Figure 3 compared with 101 checks done in the program in Figure 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 bounds check instruction is needed.

The new bounds checking instruction can be easily utilized to optimize the performance by a compiler. We only need to put a special prefix flag to distinguish bounds checking sentences and ordinary comparison when inserting checks into a program. For example, in Figure 3, the flag "_BP_" is associated with the variables in bounds checking sentences. In fact, this has been applied in almost every array & pointer bounds checking approaches (and is an easy revision if they don't have). In this way, a compiler can identify bounds checking sentences and translate them by using the new bounds checking instruction.

We also need to reduce the numbers of the instructions that set the lower-bound and upper-bound addresses as much as possible in order to further optimize the performance, as they are another extra code associated with array & pointer bounds checking. The bound information of a pointer is initialized when the memory address of a pointer is allocated, statically or dynamically. Usually, from C source code level, this bounds initialization for a pointer to sets the lower-bound and upper-bound addresses does not need to be done for many times in an application. It provides the foundation for optimizing the number of the lower-bound and upper-bound setting instructions in code generation. This optimization might have been done in register assignment optimization with traditional compiler optimization. With the special concern for bounds setting instructions, an additional global optimization can be deployed with specific registers to contain bounds information in code generation.

4 Optimization with Special Bound Checking Instruction

In this section, we propose a new instruction to perform bound checking in order to reduce the execution time of one check. We first analyze the execution time of performing one bound checking. Then we design a new instruction called BCK to perform bound 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 [10] 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 bound check. Assume that the object address, the lower-bound address and the upper-bound address are in register R1, R2, R3, respectively. Figure 4 shows the execution to perform one bound check on DLX architecture, in which instruction *"SLT R1, R2, R4"* means *"R4←1 if R1<R2; otherwise R4←0"*, and instruction *"BNER R4,Out_Bounds"* means *"jump to Out_Bounds if R4 ≠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 bound check-

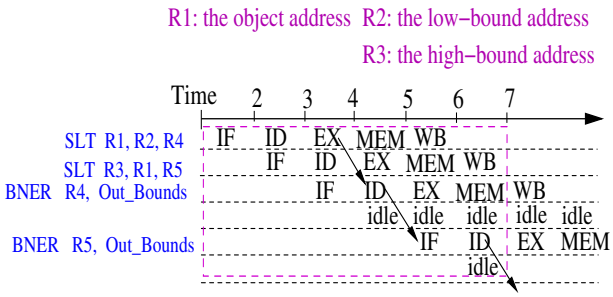

Figure 4. It takes six clock cycles to perform one bound check in the DLX architecture.

ing. 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 bound 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 bound checking can be dealt by 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 bound checking instruction called BCK in the DLX architecture has the format and meaning as shown in Table 1.

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

Table 1. The new bound checking instruction.

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 can not cause any memory/register change, as it is at the end of ID phase at that time. As shown in Figure 5, 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.

In Intel 80x86, there is an array index checking instruction that can do checking for array indexes similar to our proposed instruction in DLX architecture. Our performance

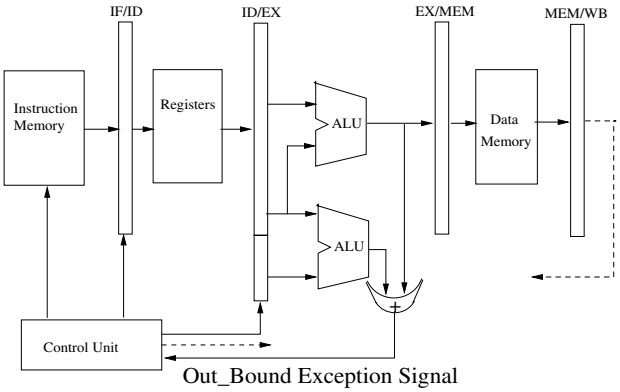

Figure 5. The implementation of the special bound checking instruction in DLX.

analysis on DLX architecture shows that a big performance improvement can be achieved if this kind of bound checking instruction is applied in optimization. The extra instructions of setting bound information for a check can usually be hoisted out of loops, and may only need to assign very few times with an additional global optimization in an application. So in the DLX architecture, it achieves about 83.3% reduction in the execution time and about 75% in the number of instructions for one check.

5 Experiments

In this section, we experiment with our array & pointer bound checking method on a small set of programs. In order to obtain cycle-accurate measurement, we use the SimpleScalar/ARM Simulator [8] 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 Figure 2. FFT (Fast Fourier Transform), IFFT (Inversed FFT) and String Search are selected from MiBench, a free, commercially representative embedded benchmark suite[6]. 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).

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 bound checking statements, we apply our checking policy that only checks the deference associated with "write" operations. A protected program without optimization is a program in which our array & pointer bound checking approach in Section 2 is directly applied. A protected program with optimization is the program using the

Benchmarks	Original	Protected without Optimization		Protected with Optimization	
	time (cycles)	time (cycles)	Overhead (%)	time (cycles)	Overhead (%)
The Example (Figure 2)	37349	39022	4.48%	37711	0.97%
FFT (small)	78696891	81596639	3.68%	79164981	0.59%
FFT (large)	471268795	912067050	93.53%	477295365	1.28%
IFFT (small)	137057269	140622324	2.60%	140622282	2.60%
IFFT (large)	472513514	716907862	51.72%	478710497	1.31%
String Search	5402119	9752968	80.54%	5970558	10.52%
MatMpy (small)	1495814	1596768	6.75%	1517411	1.44%
MatMpy (large)	278454909	389366176	39.83%	315239739	13.21%
QuickSort (small)	18715285	20016684	6.95%	19867346	6.16%
QuickSort (large)	196385271	212044087	7.97%	210545337	7.21%
Average Overhead over Original Time.			29.81%		4.53%

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

software optimization method. The experimental results are shown in Table 2.

From Table 2, we can see that the overhead is small. For programs protected by bound checking without optimization, the average overhead is 29.81%. Compared with the previous work that has 2-5 times overhead, the improvement comes from our write-only checking policy. The average overhead is 4.53% with the optimization. In the experiments, we did not apply special bound checking instruction in the optimization. We expect further performance optimization by combining special bound checking instruction into compilers in our future work. With such low overhead, array & pointer bound check can be a practical solution to defend against buffer overflow attacks.

6 Conclusions

Array & pointer bound checking is one of the most effective approaches for defending against buffer overflow attacks. However, original array & pointer bound checking causes too much overhead since it is designed to catch memory errors and it puts too many checks. In this paper, we propose an efficient array & pointer bound checking strategy to defend against buffer overflow attacks by only checking the bounds of write operations. The experimental results show that our strategy can greatly reduce the overhead of array & pointer bound checking with the optimization via software/hardware. Our conclusion is that based on our strategy, array & pointer bound checking can be a practical solution for defending systems against buffer overflow attacks with tolerable overhead.

References

[1] T. M. Austin, E. B. Scott, and 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*, pages 290–301, June 1994.

[2] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer-overflow vulnerabilities. In *Proc. of the USENIX Security Symposium*, August 2003.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grie, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the USENIX Security Symposium*, January 1998.

[4] G. Fink, C. Ko, M. Archer, and K. Levitt. Towards a property-based testing environment with applications to security-critical software. In *Proceedings of the 4th Irvine Software Symposium*, 1994.

[5] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[7] S. C. Kendall. Bcc: Runtime checking for c programs. In *Proceedings of the Summer USENIX Conference*, 1983.

[8] S. LLC. *SimpleScalor/ARM*. World Wide Web, http://www.eecs.umich.edu/ taustin/code/arm/simplesim-arm-0.2.tar.gz.

[9] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. In *the 2nd International Workshop on Automated and Algorithmic Debugging*, May 1995.

[10] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, 1996.

[11] Z. Shao, Q. Zhuge, Y. He, and E. H.-M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *IEEE 19th Annual Computer Security Applications Conference*, pages 352–361, Las Vegas, Dec. 2003.

[12] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.