Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks *

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

With more embedded systems networked, it becomes an important research problem to effectively defend embedded systems against buffer overflow attacks and efficiently check if systems have been protected. In this paper, we propose the HSDefender (Hardware/Software Defender) technique that considers the protection and checking together to solve this problem. Our basic idea is to design a secure instruction set and require third-party software developers to use secure instructions to call functions. Then the security checking can be easily performed by system integrators even without the knowledge of the source code. We first classify buffer overflow attacks into two categories, stack smashing attacks and function pointer attacks, and then provide two corresponding defending strategies. We analyze the HSDefender technique in respect of hardware cost, security, and performance, and experiment with it on the SimpleScalar/ARM simulator using benchmarks from MiBench. The results show that HSDefender can defend a system against more types of buffer overflow attacks with less overhead compared with the previous work.

1 Introduction

Buffer overflow attacks are one of the most serious security threats. More than 50% of today's widely exploited vulnerabilities are caused by buffer overflow and the ratio is increasing over time. In 2003, buffer overflows account for 67.9% (19 of 28) of the serious vulnerability reports from CERT advisories. Buffer overflow attacks cause serious security problems to special purpose embedded systems as well as general purpose systems. With more embedded systems networked, it becomes an important research problem to defend embedded systems against buffer overflow attacks.

Due to the increasing complexity of embedded applications and the strict requirements in latency, throughout, power consumption, area, cost, etc., it becomes more attractive and necessary to integrate as many off-the-shelf components as possible in embedded system designs. Therefore, a typical design needs to go through three phases: (1) System integrators assign tasks to third-party software developers with certain requirements and rules. (2) Third-party software developers generate software components based on the requirements and rules. (3) System integrators check whether all rules have been followed and all requirements have been satisfied. Considering this design flow, an effective solution to protect embedded systems against buffer overflow attacks must contain two factors:

- It must provide a complete protection and the requirements and rules must be simple so third-party software developers can easily follow.
- It must provide an efficient checking mechanism so system integrators can easily check whether a component has been protected or not. Since the source code of some components may not be available for system integrators, the security checking must be able to be performed even without the knowledge of source code.

These two factor are considered separately in most previous work. Some approaches have been developed to protect a system against buffer overflow attacks. Runtime boundary checking [1] adds instructions to check array bounds and performs pointer checking in run time. While using this strategy may completely protect a general system against buffer overflow attacks, a big performance overhead may occur especially for array and pointer intensive applications. StackGuard [4],etc., are proposed to defend against stack smashing attacks. PointGuard [3], Address Obfuscation [2], etc., are proposed to defend against pointer-corruption attacks. The combination of these techniques may be useful for the protection of general systems. However, using these techniques in embedded system integration, it is very hard for system integrators to check whether a software component has been protected or not based on binary code. For critical embedded system applications, system crash must be avoided. But if PointGuard or Address Obfuscation is used, it might be easy to cause system crashes when pointer-corruption attacks arrive.

Some approaches have been developed to check buffer overflow vulnerabilities. The static checking method uses the strategy to detect the vulnerabilities by analyzing the C source code [11] using software tools. The protection provided by this method may be incomplete and imprecise, as only known vulnerabilities can be detected and the general buffer overflow detection problem is undecidable. Program testing strategy [6] checks buffer overflow vulnerabilities by executing programs with specific inputs. While using this strategy can catch most vulnerabilities, the protection may not be complete because the detection depends on the test data to cause overflows.

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001 and NSF CCR-0309461, USA.

In [10], we propose an approach that considers protection and checking together to protect embedded systems. However, in the approach, the stack overflow protection technique has some requirements that may limit its applicability and the function pointer protection technique may cause the system crash.

In this paper, we propose a new approach, called HSDefender (Hardware/Software Defender), to protect embedded systems against buffer overflow attacks. Since hardware/software codesign such as hardware modification or adding new instructions is common practice in embedded systems design, our basic idea is to design a secure instruction set and require third-party software developers to use secure instructions to call functions. Then a system integrator can easily check whether old call instructions are used in a component or not even without the knowledge of the source code. We classify overflow-based attacks into two categories: *stack smashing attacks* and *function pointer attacks*. HSDefender approaches each of them with different mechanisms. HSDefender achieves the following properties:

- For *stack smashing attacks*, the most common attacks, we propose two methods to completely protect a system against these attacks and avoid the system crash (See Section 2.1).
- For *function pointer attacks*, our method will make it extremely difficult for a hacker to change a function pointer leading to the hostile code. Using the idea to compare a flag instruction before jumping, our method can effectively avoid the system crash (See Section 2.2).
- The requirements and rules are easy to be followed by software developers, and the security checking can be efficiently performed by system integrator even without the knowledge of the source code.
- The overhead is minimum. There is only a very minor hardware addition to a CPU architectures. The performance overhead is also minimal so HSDefender can be used to protect real-time embedded systems.

We give a case study based on four classes of processors to show how the *HSDefender* technique can effectively protect an embedded system against buffer overflow attacks and easily check the protection even based on binary code. We analyze and experiment with our HSDefender technique on the SimpleScalar/ARM Simulator [7]. The results show that *HSDefender* can defend a system against more types of buffer overflow attacks with much less overhead compared with the previous work.

The rest of this paper is organized as follows. In Section 2, our Hardware/Software defending technique, HSDefender, is presented. Section 3 gives a case study for the implementation of HSDefender on various processors. The performance comparison and experiments are presented in Section 4 and Section 5, respectively. Section 6 concludes this paper.

2 Hardware/Software Defender

In this section, we first classify buffer overflow attacks into two categories. Then we propose and analyze our Hardware/Software Defender (HSDefender) technique, in which two corresponding defending strategies are proposed.

Overflow-based attacks can be divided into two categories: *stack smashing attacks* and *function pointer attacks*. *Stack smashing attacks* either overwrite a return address or overwrites a frame pointer, which indirectly changes a return address when the caller function returns. This is the easiest attack for a hacker to do, and also the most common one. *Function pointer attacks* has two types: *local function pointer attacks* and *shared function pointer attacks*. *Local pointer attacks* exploit a local function pointer that is a pointer pointing to a function. Its value is determined by the linker when an executable file is generated during linking. A local function pointer can be exploited either directly or indirectly [9] *Shared function pointer attacks* exploit a shared function pointer that is a pointer pointing to a shared function. Its value is determined by the dynamic linker when a shared function is loaded into the memory during running time. Since it is common that a program is dynamically linked with library functions, there is usually a table storing shared function pointers in a program image. *A shared function pointer attack* activates the attack code by exploiting a shared function pointer in this table [9]. *Function pointer attacks* are not as easy to exploit as stack smashing attacks.

Two components, *stack smashing protection* and *function pointer protection*, are proposed in HSDefender to defend against these two kinds of attacks, respectively.

2.1 Component 1: Stack Smashing Protection

Two methods, *hardware boundary check* and *secure function call*, are proposed to protect from stack smashing attacks.

Method 1: Hardware Boundary Check The protection scheme is to perform *hardware boundary check* using the current value of the frame pointer. The basic approach is: (1) While a "write" operation is executed, an "address check" is parallel performed for the target's address. (2) If the target's address is equal to or bigger than the value of the frame pointer, the stack overflow exception is issued; otherwise, do nothing.

The requirement for third-party software developers is: if a variable needs to be changed in child function calls, the variable should be defined as a global variable, static variable, or dynamic memory allocation (in data, BSS, heap segment) rather than a local variable (in the stack).

The security check is very easy. A system integrator can execute the tested program and see whether the stack overflow exception occurs. In runtime, the system crash can be avoided by calling recovery program in the stack overflow exception handler program.

The advantages of this approach are as follows: (1) A system can be completely protected from *stack smashing attacks* since all the frame pointer, return address and arguments are protected. (2) The writing and the boundary checking can be executed parallel; therefore, there is no performance overhead. (3) The source code and the extra protection code are not needed.

Method 2: Secure Function Call In this method, we design two secure function call instructions: "SCALL" and "SRET". Basically, "SCALL" will generate a signature of the return address when a function is called, and "SRET" will check the signature before returning from this function. We require that third-party software developers must use these secure function calls instead of the original ones when calling a function.

To implement "SCALL" and "SRET", each process is randomly assigned a *key* when it is generated and the *key* is kept in a special register $\mathtt{R}$. "SCALL" is used to replace the original "CALL" instruction. Basically, a "CALL" instruction has 2 operations: push the return address into the stack and then put the address of the function into *Program Counter* to execute the function. "SCALL" adds the operations to generate the signature. It has four operations: (1) Push the return address into the stacks; (2) Generate a signature $\mathtt{S}$ by $\mathtt{S} = \mathtt{XOR}(\mathtt{R}, \mathtt{Ret})$, where $\mathtt{R}$ stores the key and $\mathtt{Ret}$ is the return address; (3) Push $\mathtt{S}$ into the stack; (4) Put the address of function into *Program Counter*.

"SRET" is used to replace the original "RET" instruction. Basically, a "RET" instruction will pop the current two values in the stack to the frame pointer and the Program Counter. Let (SP) denote the value in the location pointing by SP. "SRET" adds the operations to check the signature. It has four operations: (1) Load (SP) and $(SP+4)$ to two temporary registers, T_1 and T_2 (T_1 and T_2 store the signature and the return address pushed in $SCALL$, respectively); (2) Calculate $S' = XOR(R, T_2)$. (3) Compare T_1 and S': if equal, move T_2 to the Program Counter; otherwise, generate a stack overflow exception.

If the return address is changed by a hacker, it can be found since two signatures (S' and S) are different. Since the *key* is randomly generated for each process, it is extremely hard for a hacker to guess the *key*. So a hacker can not give a correct signature if he changes both the return address and the signature.

A system integrator can easily check whether there are the original "CALL" instructions in a component based on binary code and execute it to see whether there is stack overflow exception. In runtime, the system crash can be avoided by calling a recovery program in the exception handler program.

Compared with method 1, this method introduces more performance overhead and hardware cost. We give the comparison for these two methods on hardware cost and performance overhead in Section 5.

2.2 Component 2: Function Pointer Protection

To protect function pointers from being exploited, we add a new instruction called "SJMP" and require third-party software developers to use it if calling a function by function pointers.

As in method 2 in stack smashing protection, each process is randomly assigned a *key* when it is generated and the *key* is stored in a special register R. The operations of $SJMP$ are: (1) XOR the input address with R (the *key*); (2) Compare the instruction in the XORed address with a flag instruction: if they are equal, jump to this XORed address; otherwise, issue a buffer overflow exception. The value of the flag instruction is a special instruction and we can identify if it is the beginning of a function based on it. The first instruction in a function call is usually special and rarely used in other locations in a program. Therefore, we can use it as a flag instruction. For example, for Intel X86, it is "push %bp"; for Sun Sparc, it is "save".

Our *function pointer protection* technique has two requirements for third party software developers: (1) When they assign the address of a function to a function pointer, the address of the function is first XORed with R (the *key*) and then the result is put into a function pointer. (2) When they call functions using function pointers, they must use the secure jump instruction "$SJMP$".

A system integrator can easily check whether $SJMP$ has been used in all function calls that use function pointers based on binary code. In a program, there are two forms that a function call through a function pointer is translated: calling indirectly through a memory location or a register where the address of a function is stored. Therefore, a system integrator can easily check whether there are such indirect function calls based on binary code. An example are shown in Section 3.

If a hacker changes a function pointer and makes it point to the attack code, the attack code can not be activated because the real address that the program will jump to is the XORed address with the key. To compare the instruction in the XORed address with the flag can avoid the system crash in most cases. The *key* is stored in a special register, therefore, the key value cannot be overwritten by buffer overflow attacks. Since the *key* is randomly generated for each process, it is extremely hard for a hacker to guess the *key*. Thus, our method can defend a system against function pointer attacks.

3 Application of HSDefender

In Section 2, we present the general technique of HSDefender. In this section, we will show how to apply HSDefender on various processors. FFT (Fast Fourier Transfer), one of the most widely used application in Digital Signal Processing, is used as an exemplary application. We have applied our *HSDefender* technique on four classes of processors: Intel x86 processors, ARM processors, TI's TMS320C5400 processors, and Sun Sparc processors. Due to the limitation of the space, we only put the application on Intel x86 processors in this paper.

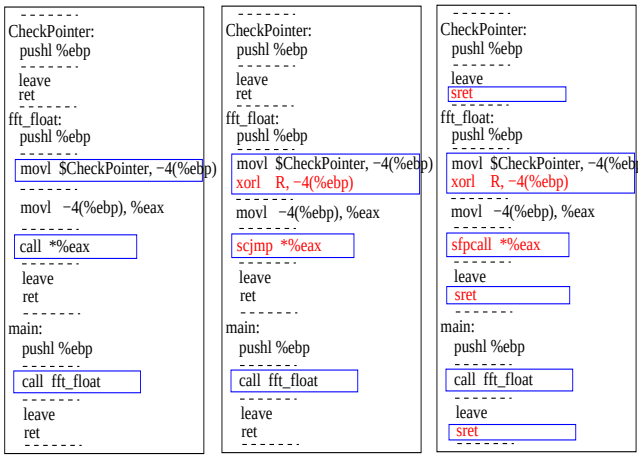

Figure 1. The assembly programs of FFT/IFFT on Intel-x86-like processors.

The program for FFT we use is the FFT/IFFT benchmark from MiBench, a free, commercially representative embedded benchmark suit [5]. Three are three functions: main(), fft_float(), and CheckPointer() in the FFT/IFFT benchmark. The basic calling procedure is: main() first prepares the inputs and then calls fft_float(); fft_float() performs the fourier transfer and needs to call CheckPointer(). The original program has no function pointer. In order to show how to defend function pointer attack in HSDefender, we add a function pointer,`fptr`, in fft_float() and use it to call CheckPointer() in fft_float().

The corresponding assembly problems of FFT/IFFT on Intel-x86-like processors are shown in Figure 1, in which Figure 1(a) shows the assembly program without any protection, and Figure 1(b) and (c) show the assembly programs with the HSDefender protection using different stack smashing protection methods. In the assembly program in Figure 1(b), hardware boundary check method is used to protect against stack attacks. Since arguments, return address and the previous frame pointer are stored below fp in Intel-x86-like processors, they are protected if a write operation is denied when its address is equal to or bigger than fp. Therefore, we can directly implement *stack smashing protection* of HSDefender technique on Intel-x86-like processors by parallel checking the address of a write. The check is automatically done by hardware when performing "write" operation, so the source code does not need to be revised. Therefore, we don't need to change any "call" or "ret" instructions except "call * %ax" that uses function pointer as shown in Figure 1(b).

In the assembly program in Figure 1(c), *secure function call* method is used to protect against stack smashing attacks. Using

this method, a third-party software developer is required to use the secure function call instructions, “scall” and “sret”, to replace the original “call” and “ret” in a program. Therefore, all “call” and “ret” (except “call * %ax”) in the original assembly program in Figure 1(a) are changed to “scall” and “sret” correspondingly in Figure 1(c).

To protect against function pointer attacks, a third-party software first needs to store the XORed address (obtained by XORing the address of a function with the key in register R) to a function pointer rather than the address of a function. Thus, the instruction “xorl R, -4(%ebp)” is added after “movl $CheckPointer,-4(%ebp)” in Figure 1(b) and (c). Then a third-party software needs to use “SJMP” to replace the call using function pointer. We need to combine “SJMP” with “call” since the indirect call using a register such as ”call *%eax” is used for a fuction call through a function pointer on Intel-x86-like processor. Considering the different methods used in stack smashing protection, there are two different forms to replace “call *%eax” with the combination of “SJMP” and “call”. If *hardware boundary check* is used in stack smashing protection, “scjmp” is used. The operations of “scjmp” are: (1) Push the return address into the stack; (2) Using “SJMP %eax” to jump to the function. If *secure function call* is used in stack smashing protection, “sfpcall” is used. The operations of “sfpcall” are: (1) Get the XORed address by XORing the address stored in %eax with R; (2) Check if the instruction stored in the XORed address equals the flag instruction “push %ebp”; (3) If equal, using “scall XORed_Address” to call the function; otherwise, issue the buffer overflow exception. Therefore, in Figure 1(b) and (c), “call *%eax” is replaced by “scjmp *%eax” and “sfpcall *%eax”, respectively.

From the assembly programs protected by HSDefender in Figure 1(b)-(c), we can see a system integrator can easily check whether a program has been protected. For a program using *hardware boundary check* method, if the indirect call or jump instruction using “call” exists in a program, it is not protected. For a program using *secure function call* method, if there exists “call” instruction, it is not secure.

4 Comparison and Analysis

In this section, we first compare and analyze the two methods for *Stack Smashing Protection*, the first component of HSDefender (Section 2.1). Then we discuss hardware cost and time performance of *Function Pointer Protection*, the second component of HSDefender.

4.1 The Comparison of The Two Methods

The two methods of stack smashing protection of HSDefender, *hardware boundary check* and *secure function call*, are compared in respect of security, hardware cost and time performance.

Security. Both methods guarantee that it is extremely hard for a hacker to use stack smashing attacks to execute the inserted hostile code. However, *hardware boundary check* provides a even better security than *secure function call*, because *hardware boundary check* method protects frame pointers, return addresses and arguments while *secure function call* only protects the return address.

Hardware Cost. Both methods need very simple hardware. *Hardware boundary check* only needs a comparator that can compare if the target’s address of a write operation is equal to or greater than the value of fp. *Secure function call* needs two components: a XOR-operation unit that XORs the return address and the key to generate a signature, and a comparator to compare if two register values are equal. To have a realistic comparison, we designed and synthesized the hardware based on the two methods assuming 16-bit word length. We describe our hardware designs by VHDL in RTL (Register Transfer Level) and perform simulation and synthesis using Synopsis. The hardware architectures for the comparators for hardware boundary check and secure function call are shown in Figure 2(a) and (b), respectively. The synthesis results are shown in Table 1.

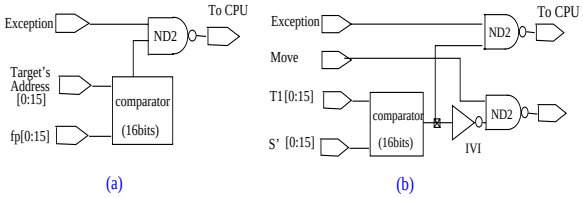

Figure 2. The comparators.

Hardware boundary check has less hardware cost compared with secure function call. The overall time performance is discussed next.

Method	Component	The number of gates	Delay (ns)
Hardware Boundary Check	Comparator	122	15.17
Secure Function Call	Comparator	125	15.17
	XOR-operation Unit	144	4.29

Table 1. The synthesis results by Synopsis.

Time Performance. As most modern CPUs are designed with pipelining, both methods introduce very little overhead. To give a realistic comparison, we analyze time overhead of the two methods based on the five-phase pipeline architecture of DLX [8]. The five phases are: IF (instruction fetch), ID (instruction decode), EX (execution), MEM (memory access), WB (write back). To make a fair comparison, the worst case is considered.

Using *Hardware Boundary Check*, we compare the target’s address with the value of fp for each write operation. We can add the additional hardware to parallel perform the comparison while the write operation can be performed at the same time in MEM phase. The buffer overflow exception is issued only when the address is greater than or equal to the value of fp; otherwise, nothing happens. Therefore, there is no overhead.

Using *Secure Function Call*, “SCALL” adds two more operations to the original “CALL”: one is to generate the signature and the other is to push the signature into the stack. Considering the worst case, it needs two extra clock cycles to finish, although it is possible to overlap the first operation (generate the signature by XORing) with the pipeline stalls caused by “CALL”. “SRET” adds three more operations to the original “RET”. So in the worst case, three extra clock cycles are added for each ”SRET”. Totally, there are at most 5 extra clock cycles for each call.

Recommendations. Based on the above analysis, if security and performance are the major concerns, hardware boundary check is recommended, which needs simpler hardware and does not introduce overhead. If applicability is the major concern, secure function call is recommended. The performance experiments

in Section 5 show that the average overhead for MiBench on a StrongARM simulator is small for secure function call method.

4.2 Analysis for Component 2 of HSDefender

Function pointer protection, component 2 of HSDefender, needs to decrypt a function pointer by XORing the address of a function with the key. Its "SJMP" instruction needs to add three more operations to "JMP": (1) XOR the given address with the key; (2) Load the instruction in the XORed address into a temporary register; (3) Compare if it is equal to the flag instruction. In the worst case, the extra clock cycles for each call through function pointer is 4 clock cycles. Since function pointer protection uses the similar operations as those in secure function call method, it needs the similar hardware component as that for secure function call method.

5 Experiments

In this section, we experiment with our HSDefender technique on the benchmarks from MiBench. As the analysis in Section 4, if hardware boundary check method is used as stack smashing protection, there is no performance overhead. Function pointers are rarely used in embedded applications; therefore, the total overhead approximates 0 if hardware boundary check is used. So in this section, we only consider the case that secure function call method is used as stack smashing protection.

Benchmarks	Total Clock Cycles		Overhead
	The Original Program	The Protected Program	%
FFT	18640705	18644716	0.021
CRC32	2316859108	2316859414	0.0000132
Susan(smooth)	81385864	81387446	0.00194
Susan(edges)	7346473	7347681	0.016
Susan(corners)	3245126	3245832	0.022
qsort	246270661	246835672	0.229
stringsearch	543606	543675	0.013
sha	44482645	44531664	0.110
rijndael(encrypt)	73410009	73538437	0.175
rijndael(decrypt)	72148502	72319177	0.237
Dijsktra	173749902	173896780	0.085
Average Overhead			0.083

Table 2. The time overhead.

The SimpleScalar/ARM Simulator [7] that is configured as the StrongARM-110 microprocessor architecture is used as the test platform. Various benchmarks are selected from MiBench and compiled as ARM-elf executables using a GNU ARM-elf cross compiler.The pipeline architecture of a StrongARM-110 microprocessor is similar to that of DLX, which we use as the exemplary architecture when analyzing the performance in Section 4. For each benchmark, we add the corresponding overhead into the assembly code obtained by the cross compiler and then generate the simulated protected executable. The protected executables and the original ones are executed into the SimpleScalar/ARM simulator and the total clock cycles are recorded and compared. The test results are shown in Table 2.

From Table 2, the results show that there is very little overhead caused by HSDefender. The average overhead is 0.083%. Therefore, HSDefender can be used to protect real-time embedded systems.

6 Conclusion

In this paper, we proposed the Hardware/Software Defending Technique (HSDefender) to defend embedded systems against buffer overflow attacks. Considering protection and checking together, HSDefender provides a mechanism that can effectively protect embedded systems against buffer overflow attacks and efficiently check if a component has been protected even without presence of source code. We classified buffer overflow attacks into two categories and then provided two corresponding defending components. We showed that our HSDefender technique can be applied to various processors. We analyzed and experimented our HSDefender technique with MiBench, a free and commercially representative embedded benchmark suite, on the SimpleScalar/ARM simulator. The results show that our HSDefender technique can defend more types of buffer overflow attacks with much less overhead compared with the previous work.

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] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[3] 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.

[4] 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.

[5] 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.

[6] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2003.

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

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

[9] G. Richarte. *Four different tricks to bypass stackshield and stackguard protection*. World Wide Web, http://www1.corest.com/files/files/11/StackGuardPaper.pdf, April 2002.

[10] 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.

[11] 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.