Security Protection and Checking for Embedded System Integration against Buffer Overflow Attacks via Hardware/Software

Zili Shao, *Member*, *IEEE*, Chun Xue, *Student Member*, *IEEE*,
Qingfeng Zhuge, *Student Member*, *IEEE*, Meikang Qiu, *Student Member*, *IEEE*,
Bin Xiao, *Member*, *IEEE*, and Edwin H.-M. Sha, *Senior Member*, *IEEE*

Abstract—With more embedded systems networked, it becomes an important problem to effectively defend embedded systems against buffer overflow attacks. Due to the increasing complexity and strict requirements, off-the-shelf software components are widely used in embedded systems, especially for military and other critical applications. Therefore, in addition to effective protection, we also need to provide an approach for system integrators to efficiently check whether software components have been protected. In this paper, we propose the HSDefender (Hardware/Software Defender) technique to perform protection and checking together. Our basic idea is to design secure call instructions so systems can be secured and checking can be easily performed. In the paper, we classify buffer overflow attacks into two categories and provide two corresponding defending strategies. We analyze the HSDefender technique with respect to hardware cost, security, and performance. We experiment with our HSDefender technique on the SimpleScalar/ARM simulator with benchmarks from MiBench, an embedded benchmark suite. The results show that our HSDefender technique can defend a system against more types of buffer overflow attacks with less overhead compared with the previous work.

Index Terms—Security, buffer overflow attack, embedded system, hardware/software, protection.

1 Introduction

It is known that buffer overflow attacks have been causing serious security problems for decades. More than 50 percent of today's widely exploited vulnerabilities are caused by buffer overflow and the ratio is increasing over time. One of the most famous examples in the early days is the *Internet worm* in 1988 that made use of buffer overflow vulnerabilities in *fingerd* and infected thousands of computers [2]. Recent examples include the *Code Red*, *Code Red II*, and their variations which exploited known buffer overflow vulnerabilities in the Microsoft Index Service DLL. 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. In 2003, buffer overflows accounted for 70.4 percent (19 of 27) of the serious vulnerability reports from CERT advisories.

Buffer overflow attacks cause serious damage to special purpose embedded systems as well as general purpose systems. Because of the growing deployment of networked embedded systems, security has become one of the most significant issues for embedded systems. Many special purpose embedded systems are used in military and critical commercial applications. For example, a battle ship or an aircraft has thousands of embedded components and a nuclear plant has numerous networked embedded controllers. A hostile penetration by using buffer overflow attacks in such facilities could cause dramatic damage. In this paper, we address the security issues of special purpose embedded systems caused by buffer overflow attacks.

Due to the increasing complexity of embedded applications and the strict requirements in latency, throughput, power consumption, area, cost, etc., it becomes more attractive and necessary to design an embedded system by integrating as many off-the-shelf components as possible. Considering this trend, from a system integrator's point of view, there are three phases in the design flow, as shown in Fig. 1:

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 the security problems that are caused by buffer overflow attacks, in Phase 1, system integrators set security requirements and rules; in Phase 2, third-party developers add buffer overflow attack protection based on the requirements and rules; in Phase 3, system integrators check whether components have been protected from buffer overflow attacks. Therefore, to effectively defend an embedded system against buffer overflow attacks, we must provide an approach that includes both the protection and checking steps. The source code of some components may

- *Z. Shao and B. Xiao are with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: {cszlshao, csbxiao}@comp.polyu.edu.hk.*
- *C. Xue, Q. Zhuge, M. Qiu, and E.H.-M. Sha are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083. E-mail: {cxx016000, qfzhuge, mxq012100, edsha}@utdallas.edu.*

Manuscript received 7 Sept. 2004; revised 21 July 2005; accepted 14 Sept. 2005; published online 22 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0291-0904.

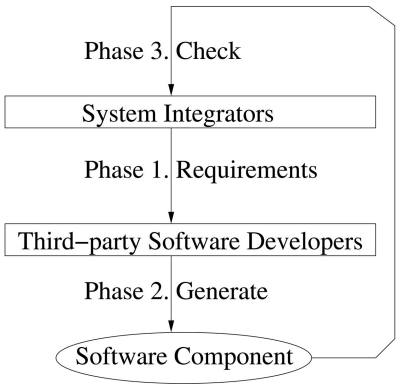


Fig. 1. The basic design flow in embedded systems using components.

not be available when performing security checking, which also needs to be considered.

A common approach to solving this problem is to ask programmers to always do boundary checks. However, it is not realistic to assume that all programmers will follow this good practice or to assume that every off-the-shelf software is immune to buffer overflow attacks. The use of the safe programming languages is an effective way to defend against buffer overflow attacks. But, for embedded system applications, most software is still written in "unsafe" languages such as C or assembly.

A lot of techniques have been proposed to defend systems against buffer overflow attacks or check buffer overflow vulnerabilities. Various techniques have been proposed to defend systems against stack smashing attacks by making the stack nonexecutable [3], intercepting vulnerable library functions and forcing verification of critical elements of stacks [4], adding extra instructions to guard systems at runtime [5], [6], and designing special hardware components and mechanism to guard the stack [7], [8], [9]. These techniques only prevent overflow attacks that overwrite return addresses along a stack. New attack techniques [10] have been developed to bypass their protection by corrupting pointers. To defend against pointer-corruption attacks, several techniques [1], [11], [12], [13], are proposed by encrypting pointer values while they are in memory and decrypting them before dereferencing.

Some techniques have been proposed to check buffer overflow vulnerabilities by adding instructions to check array bounds and perform pointer checking at runtime [14], [15], detecting the vulnerabilities by analyzing the C source code [16], and executing programs with specific inputs[17]. These techniques either incur a big performance overhead [14], [15] or can only detect known vulnerabilities [16], [17]. Our technique is based on a hardware/software method and can effectively and efficiently defend systems against not only stack smashing attacks but also function pointer attacks with minor hardware/software modifications. Such modifications may cause portability and compatibility problems for general-purpose systems. However, it is feasible for embedded systems, where hardware/software can be specially designed and optimized for a single application.

The technique, HSDefender (Hardware/Software Defender), is proposed to protect embedded systems against buffer overflow attacks and allow system integrators to easily perform security checking even without knowledge of source code. Our basic idea is to design secure call instructions so systems can be secured and checking can be easily performed. The HSDefender technique considers the protection and checking together and enables more complete protection than the previous work. In the paper, 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 system crashes.
- 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 HSDefender, the security checking is very easy for system integrators and the rules are easy to follow for software developers. With special secure call instructions, the problem of verifying whether a component is protected is transformed to the problem of checking whether there exist old call instructions. Therefore, we can use a software tool to verify components even without the presence of source code by scanning the code to find old call instructions.
- The overhead is minimal. There is only a very minor addition to the current CPU architectures. The performance overhead is also minimal, so this approach can be easily applied to embedded applications that have real-time constraints.

We analyze and experiment with our HSDefender technique on the SimpleScalar/ARM Simulator [18]. 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, examples are given to show the basic buffer overflow attack methods that provide the necessary background for understanding our approach. Our Hardware/Software defending technique, HSDefender, is presented in Section 3. The performance comparison and experiments are presented in Sections 4 and 5, respectively. Section 6 concludes this paper.

2 Background

In this section, the examples for basic buffer overflow attack methods are shown to provide the background for understanding why our approach is necessary and how it works. A common stack smashing attack example is shown first. Then, we give two advanced stack smashing attack examples. Finally, a BSS overflow attack example is given. The stack structure of Intel x86 processors is used in these examples because it is easy to explain and understand.

Fig. 2a(2) shows a typical stack structure after function *copy()* is called, where arguments, return address, previous frame pointer, and local variables are pushed onto the stack one by one. The arrows in Fig. 2a(2) show the growth

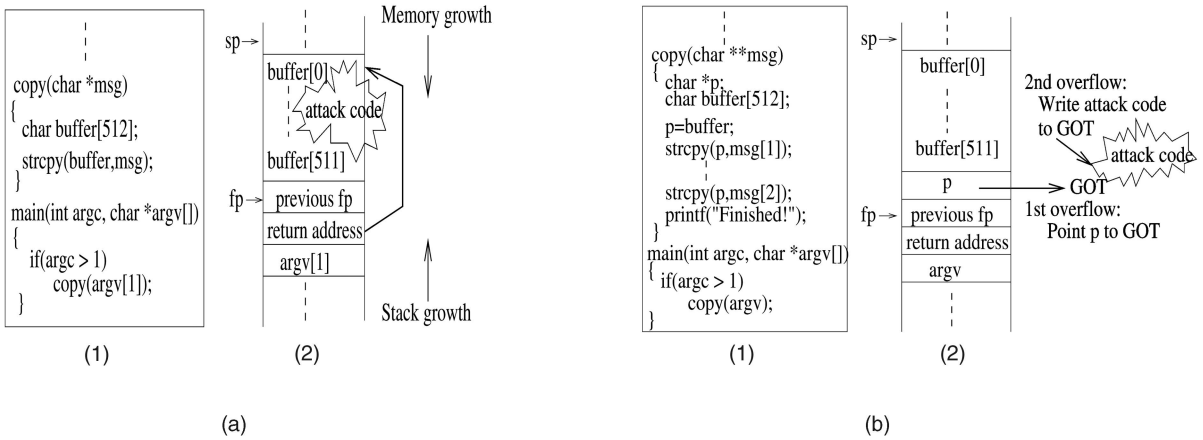


Fig. 2. (a) Vulnerable program 1 and its stack. (b) Vulnerable program 2 and its stack.

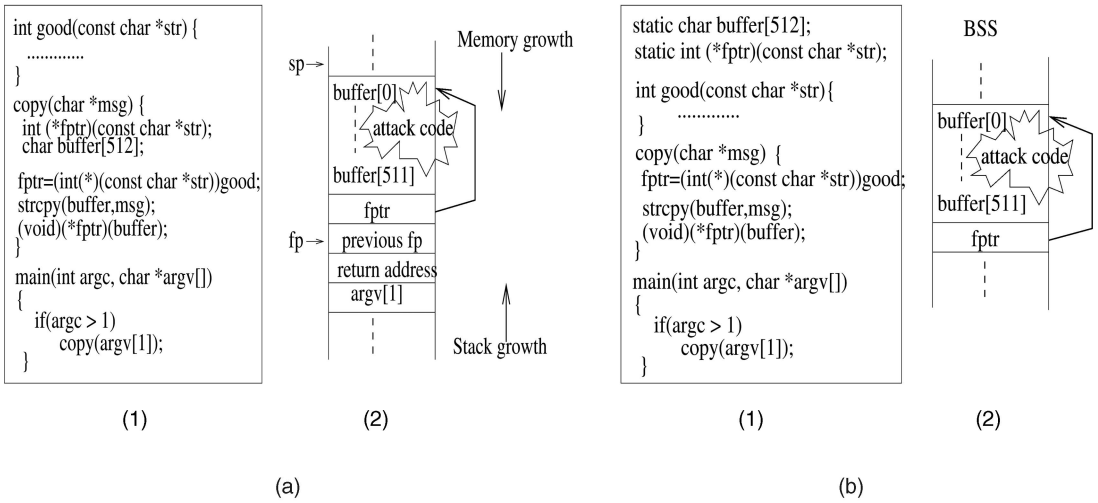


Fig. 3. (a) Vulnerable program 3 and its stack. (b) Vulnerable program 4 and its BSS (Block Storage Segment).

directions of stack and memory, respectively. Function *copy* has the most common stack overflow vulnerability. It uses *strcpy()* to copy the inputs into *buffer[]*. Since *strcpy()* does not check the size of the inputs, it may copy more than 512 characters into *buffer[]*. Therefore, the inputs can overflow the return address in the stack and make it point to the attack code injected in *buffer[]* as shown in Fig. 2a(2). Then, the attack code will be executed after the program returns from *copy()* to *main()*. Almost all techniques introduced in Section 1 can defend against this kind of attack. A variation of this kind of attacks is to overflow *previous fp* only. Since *previous fp* will point to the stack frame of *main()* after returning from *copy()*, a similar attack can be activated when the program returns from *main()*. It is used in [19], [20] to defeat the protections of StackShield and StackGuard.

Even if the return addresses and frame pointers are protected, we still cannot defend against the attack shown in Fig. 2b. The vulnerable function *copy()* in Fig. 2b(1) has one local pointer *p* and calls *strcpy()* twice. The location of *p* (Fig. 2b(2)) is below the location of *buffer[]* in the stack. The attack is deployed as follows: 1) In the first *strcpy()*, the attack code is injected into the buffer and *p* is overwritten to point to the entry of *printf* in the shared function pointer table GOT (Global Offset Table), 2) the address of the attack code is copied to the entry of *printf* in GOT by the second *strcpy()*, and 3) the attack code is activated when the program executes the call *printf("Finished!")* in *copy()*. This example shows that the shared function pointer table GOT has to be protected to defend against this kind of attack.

Our third example (Fig. 3a) shows that local function pointers might be exploited in attacks. In this vulnerable program (Fig. 3a(1)), *strcpy()* is called first and then the function pointed by function pointer *fptr* is executed in *copy()*. In the stack (Fig. 3a(2)), the location of *fptr* is below the location of *buffer[]*. Thus, *fptr* can be overwritten and pointed to the attack code by *strcpy()*. Then, the execution of *(void)(*fptr)(buffer)* will activate the attack code. Fig. 3b shows a similar example to exploit a function pointer in BSS (Block Storage Segment). In this attack, the exploitation occurs in BSS, so stack smashing protection techniques such as StackGuard, IBM SSP, etc., cannot protect against this kind of attack. These examples show that function pointers need to be protected to defend against these kinds of attacks.

3 HSDefender: Hardware/Software Defender

To effectively protect a system and efficiently perform security checking, we need to consider the protection and

the checking together. Since hardware/software codesign, such as hardware modification or adding new instructions, is common practice in embedded systems design, our basic idea is to add a secure instruction set and require third-party software developers to use these secure instructions to call functions. Then, a system integrator can easily check whether old call instructions are used in a component based on binary code. In this section, we propose our HSDefender (Hardware/Software Defender) technique. We first classify buffer overflow attacks into two categories. Then, we propose and analyze our HSDefender technique with two corresponding defending strategies.

3.1 The Categories of Buffer Overflow Attacks

It is extremely hard to design a pure hardware scheme to protect all variables in stack, heap, BSS, or any data segments from being overwritten by buffer overflow attacks. This is because, from the hardware point of view, the boundary information of each variable is not present for most contemporary hardware systems. Therefore, we should try to achieve a reasonable goal that is implementable with minimum overhead so that a secure real-time embedded system can be built.

We can classify overflow-based attacks into two categories:

Stack Smashing Attacks. This type of attacks either overwrites a return addresses as shown in Fig. 2a or overwrites a frame pointer, which indirectly changes a return address when the caller function returns. This is the easiest and also the most common attack.

Function Pointer Attacks. Based on different linking methods (static linking and dynamic linking), function pointers can be divided into two types: local function pointers and shared function pointers. Accordingly, there are two types of function pointer attacks: *local function pointer attacks* and *shared function pointer attacks*.

A local function pointer 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, as shown in Fig. 3a and Fig. 3b, or indirectly by using a method similar to that shown in Fig. 2b. If a vulnerable program has function pointers and has a particular code sequence that causes two overflows, a hacker may be able to change a local function pointer that is stored in heap, BSS, or stack.

A shared function pointer 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. For example, the shared function pointer pointing to "printf" in GOT is changed to point to the attack code in Fig. 2b.

Note that these classification categories are not mutually exclusive in some cases. For example, in a function pointer attack, a stack smashing attack may be involved in the beginning. To deploy a *function pointer attack* is harder than to deploy a stack smashing attack.

We have different levels of strength to deal with the above two types of attacks. For *stack smashing attacks*, the most common attacking methods, our goal is to completely defend against such attacks. For *function pointer attacks*, our goal is to make it extremely hard for a hacker to change a function pointer leading to the hostile code. HSDefender requires two components: **stack smashing protection** and **function pointer protection**, to achieve these goals.

3.2 Component 1: Stack Smashing Protection

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

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 performed for the target's address in parallel.
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 implementation of this hardware boundary check on different processors may be different because of different stack structures.

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). And, the frame pointer needs to be explicitly loaded in a function call.

The security check is to execute the tested program and see whether the stack overflow exception occurs. At runtime, a system crash can be avoided by calling a recovery program in the stack overflow exception handler program. The recovery program can simply terminate the program or apply other advanced methods.

The advantages of this approach are as follows: First, this scheme will guarantee that the frame pointer, return address, and arguments will not be replaced by any overflowed buffer. Therefore, we can defend a system against *stack smashing attacks* completely. Second, we can implement boundary checking in such a way that the writing and the boundary checking can be executed in parallel. Therefore, there is no performance overhead. Finally, source code and extra protection code are not needed.

This approach provides a method for system integrators to force third-party software developers to generate secure software components. However, its strict requirements may cause some difficulties for software developers. For example, a legitimate "write" operation may be denied and cause a stack overflow exception if its target address is bigger than the frame pointer. And, the functions that change environment variables may need to be revised since environment variables are located at the bottom of a program image in Unix or Linux. Therefore, a more general method is proposed next.

Method 2: Secure Function Call. While Method 1 can effectively protect systems against stack smashing attacks, its requirements are too strict. Therefore, we propose Method 2 to solve this problem. In Method 2, 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 R. The special register R can only be set up in the privilege mode through a system call. The key can be generated by a loader when it loads a program into the memory. It is unique for each process and needs to be saved and restored with each context switch in a multiprocess system. In the ordinary mode, this special register cannot be read except that it is used in "SRET" and "SCALL." "SCALL" is used to call a function instead of the original "CALL" instruction. Basically, a "CALL" instruction has two operations: push the return address onto 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 onto the stack.
2. Generate a signature S by $S = XOR(R, Ret)$, where R stores the key and Ret is the return address.
3. Push signature S onto the stack.
4. Jump to the target address (put the address of function into *Program Counter*).

"SRET" is used to return from a function instead of the original "RET" instruction. Basically, a "RET" instruction will pop the return address in the stack to the Program Counter. Let (SP) denote the value in the location pointed to 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 a return address is overflowed, it can be found since the two signatures (S' and S) are different. The *key* is randomly generated for each process, so it is extremely hard for a hacker to guess the *key*. Therefore, a hacker cannot give a correct signature even if he attempts to change both the return address and the signature.

The key is stored in a special register, so it cannot be overflowed by stack smashing attacks. But, the key is still vulnerable if an attacker has access to the protected system. Two possible vulnerabilities are as follows: **Vulnerability 1:** The key may need to be stored in the memory due to context switches. So, the key can be read from the memory (though the attacker may not need to know the key at that time since he has already gained the root privilege if he can read the key from the kernel data structure). **Vulnerability 2:** An attacker can mount a known-plaintext attack based on multiple pairs of plaintext (return address) and ciphertext (signature). Note that each process has an unique key generated randomly, so the key obtained by attacking one process cannot be applied to attack other processes.

Our method is used against stack overflow attacks that mainly attack remote services from the external of a system. Protecting against internal-user attacks is a much more difficult problem which we do not address in this work. To guess the key from the external of a system without local access, an attacker can use the following two possible methods. **Attacking Method 1:** Exhaustive search by trying all possible key values. There are two reasons that it is very hard for an attacker to find the key by the exhaustive search. First, one input value that is not the key at one time does not mean that it will definitely not be the key in the next time because the key is randomly generated when a process is constructed. Therefore, the exhaustive search by simply excluding input values from the candidate-key pool will not work. Second, trying all possible key values will cause a server to frequently be down before the case occurs in which the value matches the key. The frequent server down will warn administrators so they can find the vulnerabilities and identify the attacker, which attackers want to avoid. **Attacking Method 2:** Guess the key by constructing a similar working environment as a remote system. Since the key is randomly generated, it is very hard to make the key of a process in a local machine match the key of a similar process in the remote system. Without local access, an attacker cannot collect the information about the exact settings of the remote system, which also increases the difficulties in guessing the key. Similarly to the second reason in the exhaustive search, if this kind of attack is hard to make succeed and causes a server to frequently be down, then administrators will be warned before the attack succeeds.

From the above discussion, we can see that it is very hard to obtain the key from the outside of a system if the attacker has no access to the system.

Our method advances StackGuard in terms of code size. Code size is one of the most important concerns for embedded system designs because of very limited memory constraints. In Method 2, the generation and comparison of a signature are done in call and return instructions. In StackGuard, it needs extra instructions to push canaries onto the stack and perform comparison. Therefore, the code generated by Method 2 will need less memory space compared to StackGuard.

In terms of security checking for components, it is very easy for system integrators to check whether or not a third-party component has been protected, even based on binary code using our method. They only need to check whether the original "CALL" instructions are in a component and execute it to see whether there is a stack overflow exception. At runtime, system crashes 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 in terms of security, hardware cost, and performance overhead in Section 5.

3.3 Component 2: Function Pointer Protection

For any function call, there must be a jump operation involved. A target address of a "JMP" operation can be a constant distance or a value stored in a register or a memory

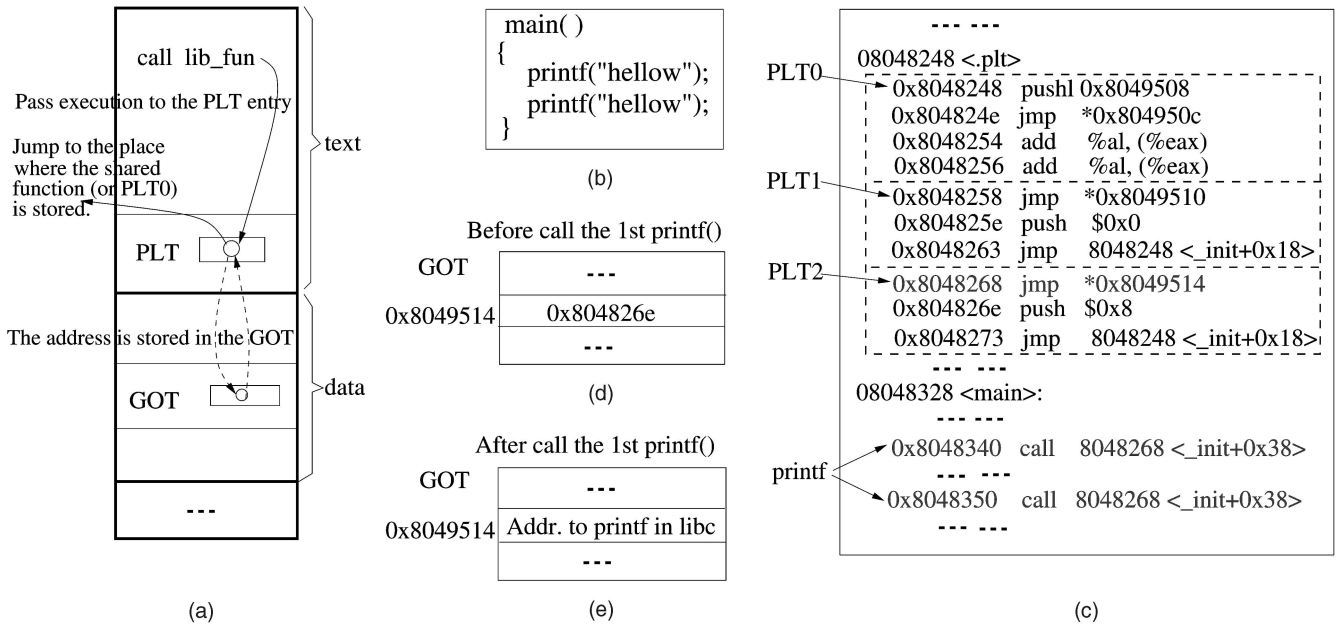


Fig. 4. (a) The execution flow of a call through shared function pointer. (b) An exemplary program. (c) The corresponding assembly code with ELF dynamic linking (elf32, Intel 386, gcc, Linux). (d) The address stored in the GOT before the first printf is called. (e) The address stored in the GOT after the 1st printf is called.

location. A function call through a function pointer must use a register or a memory location to store the target address. We call this type of jump an **indirect jump**. For any instruction involving indirect jumps, we design a new "secure" instruction with **secure jump** operations and require third-party software developers to use it whenever there is an indirect jump. The new instruction is called SJMP.

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 key is unique for each process. The operations of SJMP are:

1. Generate an address by XORing the input address with R (the *key*).
2. Jump to the XORed address.

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."

Using our function pointer protection, if a hacker changes a function pointer and makes it point to attack code, the attack code cannot be activated because the real address that the program will jump to is the XORed address with the key. 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 systems against function pointer attacks.

Our function pointer protection can be applied to protect shared function pointers. In the following, we use ELF dynamic linking to discuss this issue. In ELF dynamic linking, a typical execution flow of a call through shared library function involves two levels of indirection: PLT (Procedure Linkage Table) in text segment and GOT in data segment in a program image. As shown in Fig. 4a, when a shared library function is called, the execution is first passed to its PLT entry. (PLT is set up by the static linker at compile time. See Fig. 4c as an example.) The first instruction in the PLT entry is a indirect jump with the address stored in the GOT. The initial values of the GOT are set up by the loader when the program image is loaded into the memory. If it is the first time to call this function, then the address stored in the GOT is the address of the second instruction in the PLT entry. As shown in Fig. 4c or Fig. 5a, the second instruction in a PLT entry (except PLT0) pushes an offset and then the execution is passed to PLT0 (the zero entry). The code in PLT0 calls the dynamic linker through an address stored in the GOT (set up by the loader). The dynamic linker then finds the correct address where the shared function is stored in the memory and stores it in the GOT. When the same shared library function is called the next time, the GOT will contain the address of the function.

The above discussion is based on "lazy evaluation" that is the default mode when a program is compiled. A complete ELF specification and the techniques related to linking and loading can be found in [21], [22], [23]. Fig. 4 shows an ELF dynamic linking example. For the program shown in Fig. 4b, Fig. 4c shows the corresponding assembly code with ELF dynamic linking (obtained by using "objdump" on Redhat Linux 9). Fig. 4d and Fig. 4e show the values stored in the GOT before and after the first printf() is called, respectively.

To protect shared function pointers, we need to use our "SJMP" to replace all indirect jump instructions in the PLT, as shown in Fig. 5b, which can be done by the static linker at

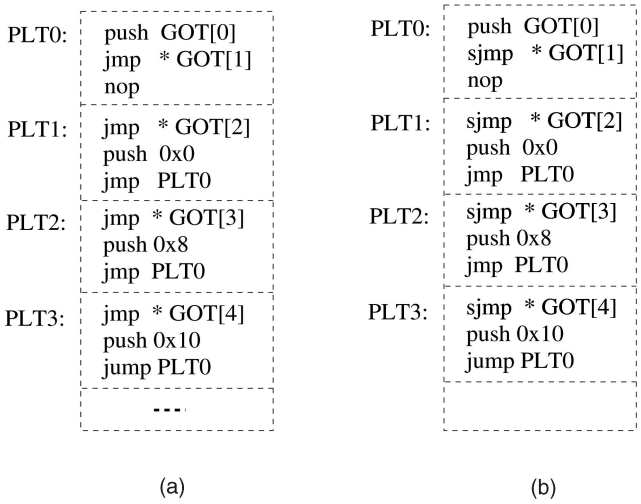


Fig. 5. (a) The PLT without protection. (b) The PLT with protection.

compile time. When loading a program, the loader needs to encrypt all initial pointers in the GOT by XORing each pointer with the key. At runtime, when resolving a symbol, the dynamic loader needs to store the encrypted address of a shared function obtained by XORing the address with the key. Each process has a key and the key is stored in a special register, so these modifications can be easily achieved by XORing with the special register.

System integrators 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 into which a function call through a function pointer is translated: either 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.

3.4 Security Analysis

The HSDefender technique protects important stack structures, shared function pointers, and local function pointers. However, it is still possible that a variable can be overwritten by a pointer in the heap or BSS. Then, an attack through a file name, like in [2], can be deployed. There is too much overhead if we protect every variable either by hardware or software. But, for important variables, such as in FILE structure, we can use software to *encrypt* them. For example, a filename or a file descriptor id can be regarded as a possible "function pointer" to the external file system. This requires a software approach to protect the integrity of any file structure. For embedded systems without file systems, the concern of such attacks is not necessary. How to protect the FILE structure from buffer overflow attacks will be one of our future research topics.

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 4.2). Then we discuss hardware cost and time performance of *Function Pointer Protection*, the second component of HSDefender (Section 4.3). Finally, we compare the protection of our HSDefender technique with other existing work.

4.1 The Comparison of the Two Methods for Stack Smashing Protection

The two methods of stack smashing protection of HSDefender, *hardware boundary check* and *secure function call*, are compared with respect to 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, as discussed in Section 3.2, *hardware boundary check* protects frame pointers, return addresses, and arguments, while *secure function call* only protects return addresses.

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: an 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 Fig. 6. The synthesis results are shown in Table 1.

From Table 1, hardware boundary check has less hardware cost compared with secure function call. The overall time performance is discussed next.

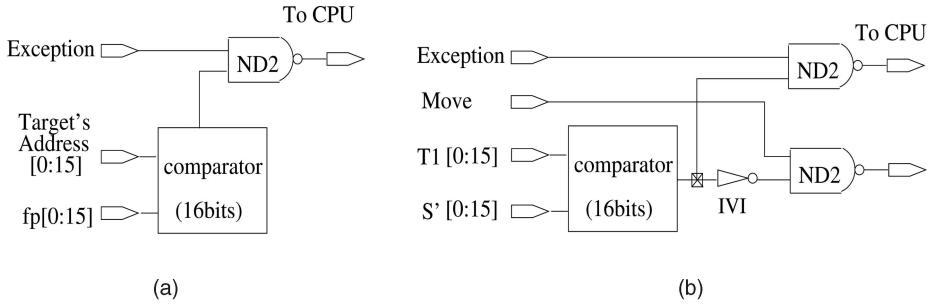


Fig. 6. The comparators (a) for hardware boundary check and (b) for secure function call.

TABLE 1
The Synthesis Results by Synopsis

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

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 [24]. The five phases are: IF (instruction fetch), ID (instruction decode), EX (execution), MEM (memory access), and WB (write back). To make a fair comparison, the worst case is considered.

Hardware Boundary Check. Using this method, we compare the target's address with the value of fp for each write operation. We can add the additional hardware to 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.

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 onto 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": 1) Load (SP) and ($SP+4$) to temporary registers T_1 and T_2, 2) generate S' by XORing T_2 with the key stored in register R, and 3) Compare S' with T_1. So, in the worst case, three extra clock cycles are added for each "SRET." In total, there are at most five extra clock cycles for each call using *secure function call* method.

The overheads introduced by the two approaches are summarized in Table 2.

Recommendations. Based on the above analysis, our recommendations are listed in Table 3. If security and performance are the major concerns, hardware boundary check is recommended, which needs simpler hardware and does not introduce overhead. If easy software implementation 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.

TABLE 2
The Overheads for Each Function Call

	Hardware Boundary Check	Secure Function Call
Overhead(clock cycles)	0	5

TABLE 3
The Recommendations Based on Different Concerns

Concerns	Recommendations
Security, Timing performance	Hardware Boundary Check
Easy software implementation	Secure Function Call

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 more operations to "JMP": XOR the given address with the key. In the worst case, the extra clock cycles for each call through function pointer is one clock cycle. Based on their similar operations, similar hardware components can be used for function pointer protection as that for secure function call.

4.3 The Comparison of Protection

Our HSDefender technique can defend against more types of buffer overflow attacks by combining its two components compared with the previous work. Using its stack smashing protection component, it can defend against stack smashing attacks no matter what return addresses or frame pointers are used. HSDefender can defend against local function pointer attacks and shared function pointer table attacks in various places, which cannot be achieved by most of the previous work. The security comparisons with previous important work are summarized in Table 4.

In the table, "No" denotes that the corresponding method listed in Column "Methods" fails to protect a vulnerable program from the corresponding attacks under field "Attacks"; "Yes" denotes that the method succeeds; "Yes/No" denotes that the method either fails or succeeds depending on certain circumstances. The "Yes/No" under column "arguments" has different meanings for IBM SSP and our HSDefender. In many cases, IBM SSP can defend against stack smashing attacks through arguments. But, it fails in some cases such as when the program has a structure that contains both a pointer and a character array. Our HSDefender can completely defend against any attacks through arguments if hardware boundary check is used in *stack smashing protection*. Although arguments are not protected if secure function call is used, it is very hard for a hacker to exploit arguments because all function pointers are protected by two components of HSDefender. HSDefender that uses secure function call can protect a system against frame pointer attacks because the change of fp can be captured through the signature comparison. PointGuard can protect the arguments if they are pointers, so we put the "Yes/No" in the column "arguments."

TABLE 4
The Security Comparison of HSDefencer with the Previous Work

Methods	**Various Buffer Overflow Attacks**								
	Stack Smashing Attacks			Shared Function Pointer Table Attacks			Local Function Pointer Attacks		
	fp	arguments	return address	stack	heap	BSS	stack	heap	BSS
HSDefender	Yes	Yes/No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
StackGuard	Yes	No	Yes	No	No	No	No	No	No
StackShield	No	No	Yes	No	No	No	No	No	No
IBM SSP	Yes	Yes/No	Yes	No	No	No	No	No	No
LibSafe	Yes	Yes	Yes	No	No	No	No	No	No
PointGuard	No	Yes/No	No	Yes	Yes	Yes	Yes	Yes	Yes

TABLE 5
The Comparison of the Execution Time of the Original Program and the Protected Program on the SimpleScalar/ARM Simulator (SA-11 Core)

Benchmarks	Total Clock Cycles		Overhead
	The Original Program	The Protected Program	%
FFT (small)	18640705	18644716	0.021
FFT (large)	399398757	399561785	0.041
CRC32	2316859108	2316859414	0.0000132
Susan(smoothing,small)	81385864	81387446	0.00194
Susan(smoothing,large)	1237939043	1237940393	0.00011
Susan(edges,small)	7346473	7347681	0.016
Susan(edges,large)	218598419	218600514	0.00096
Susan(corners,small)	3245126	3245832	0.022
Susan(corners,large)	71603528	71604760	0.00172
qsort(small)	246270661	246835672	0.229
qsort(large)	1082711652	1085927040	0.297
stringsearch(small)	543606	543675	0.013
stringsearch(large)	11890848	11903642	0.108
sha(small)	44482645	44531664	0.110
sha(large)	462392728	462902247	0.110
rijndael(encrypt,small)	73410009	73538437	0.175
rijndael(decrypt,small)	72148502	72319177	0.237
rijndael(encrypt,large)	763360206	764715394	0.178
rijndael(decrypt,large)	750165763	751891821	0.230
Dijsktra(small)	173749902	173896780	0.085
Dijsktra(large)	800337952	801634692	0.162
Average Overhead			0.097

While the combination of StackGuard and PointGuard can provide similar protection as HSDefender does, HSDefender has the following advantages:

1. The code protected by HSDefender needs less memory space compared to the code protected by StackGuard and PointGuard.
2. *HSDefender* causes much less overhead compared with the combination of StackGuard and PointGuard. Typically, it causes 5-10 percent overhead if the combination of StackGuard and PointGuard is used. As shown next, HSDefender causes about 0.1 percent overhead on average.

5 Experiments

In this section, we experiment with our HSDefender technique on the benchmarks from MiBench, a free, commercially representative embedded benchmark suite [25]. As in the analysis in Section 4, if the 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 the secure function call method is used as stack smashing protection.

The SimpleScalar/ARM Simulator [18] 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 [26]. 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. Protected executables and original ones are executed by the SimpleScalar/ARM simulator and the total clock cycles are recorded and compared. The test results are shown in Table 5.

From Table 5, the results show that there is very little overhead caused by HSDefender. The average overhead is 0.097 percent. Therefore, HSDefender can effectively protect real-time embedded systems.

6 Conclusion and Work in Progress

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 the presence of source code. We classified buffer overflow attacks into two categories and then provided two corresponding defending components. We analyzed and experimented on 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. HSDefender is more suitable for embedded system integration than the previous approaches. We are currently working on extending HSDefender so the protection can be directly enforced by hardware with less software modification.

Acknowledgments

A preliminary version of this work appeared in the *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)* [1].

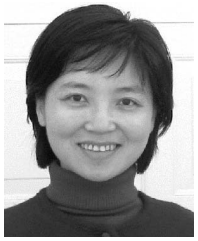
References

[1] Z. Shao, Q. Zhuge, Y. He, and E.H.-M. Sha, "Defending Embedded Systems against Buffer Overflow via Hardware/Software," *Proc. IEEE 19th Ann. Computer Security Applications Conf.,* Dec. 2003.
[2] E.H. Spafford, "The Internet Worm Program: An Analysis," Technical Report TR823, Purdue Univ., 1988.
[3] Solar Designer, "Kernel Patches from the Openwall Project," http://www.openwall.com/linux, 2002.
[4] A. Baratloo, T. Tsai, and N. Singh, "Transparent Run-Time Defense against Stack Smashing Attacks," *Proc. USENIX Ann. Technical Conf.,* June 2000.
[5] 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," *Proc. USENIX Security Symp.,* Jan. 1998.
[6] H. Etoh and K. Yoda, "Protecting from Stack-Smashing Attacks," http://ww.trl.ibm.com/projects/security/ssp/main.html, May 2002.
[7] J. Xu, Z. Kalbarczyk, S. Patel, and R.K. Iyer, "Architecture Support for Defending against Buffer Overflow Attacks," *Proc. Second Workshop Evaluating and Architecting System Dependability,* Oct. 2002.
[8] R.B. Lee, D.K. Karig, J.P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," *Proc. Int'l Conf. Security in Pervasive Computing,* Mar. 2003.
[9] H. Özdoganoglu, T.N. Vijaykumar, C.E. Brodley, and A. Jalote, "Smashguard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," Technical Report TR-ECE 03-13, Purdue Univ., Feb. 2004.
[10] Bulba and Kil3r, "Bypassing Stackguard and Stackshield," *Phrack,* vol. 5, no. 56, May 2000.
[11] Z. Shao, Q. Zhuge, and E.H.-M. Sha, "Defending Embedded Systems against Buffer Overflow via Hardware/Software (Extended Abstract)," *Proc. 2003 South Central Information Security Symp.,* Apr. 2003.
[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: Protecting Pointers from Buffer-Overflow Vulnerabilities," *Proc. USENIX Security Symp.,* Aug. 2003.
[13] S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th USENIX Security Symp.,* pp. 105-120, Aug. 2003.
[14] T.M. Austin, E.B. Scott, and S.S. Gurindar, "Efficient Detection of All Pointer and Array Access Errors," *Proc. ACM SIGPLAN '94 Conf. Programming Language Design and Implementation,* pp. 290-301, June 1994.
[15] R.W.M. Jones and P.H.J. Kelly, "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs," *Proc. Third Int'l Workshop Automated and Algorithmic Debugging,* pp. 13-26, 1997.
[16] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Network and Distributed System Security Symp.,* pp. 3-17, Feb. 2000.
[17] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proc. Network and Distributed System Security Symp.,* Feb. 2003.

[18] SimpleScalar LLC, "SimpleScalar/ARM," http://www.eecs.umich.edu/taustin/code/arm/simplesim-arm-0.2.tar.gz, 2000.
[19] klog, "The Frame Pointer Overwrite," *Phrack,* vol. 8, no. 55, Sept. 1999.
[20] G. Richarte, "Four Different Tricks to Bypass Stackshield and Stackguard Protection," http://www1.corest.com/files/files/11/StackGuardPaper.pdf, Apr. 2002.
[21] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2,* TIS Committee, May 1995.
[22] the gruqp, "Cheating the ELF: Subversive Dynamic Linking to Libraries," http://downloads.securityfocus.com/library/subversiveld.pdf, 2001.
[23] J.R. Levine, *Linkers and Loaders.* Morgan Kauffman, Oct. 1999.
[24] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.
[25] 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," *Proc. IEEE Fourth Ann. Workshop Workload Characterization,* Dec. 2001.
[26] SimpleScalar LLC, "SimpleScalar/ARM Corss Compiler Kit," http://www.eecs.umich.edu/taustin/code/arm-cross/gcc-2.95.2.tar.gz, http://www.eecs.umich.edu/taustin/code/arm-cross/binutils-2.10.tar.gz, http://www.eecs.umich.edu/taustin/code/arm-cross/glibc-2.1.3.tar.gz, 2000.

Zili Shao received the BE degree in electronic mechanics from The University of Electronic Science and Technology of China, China, 1995; he received the MS and PhD 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 codesign, and computer security. He is a member of the IEEE.

Chun Xue received the BS degree in computer science and engineering from the University of Texas at Arlington in May 1997 and the MS degree in computer science from the University of Texas at Dallas in December 2002. He is currently a computer science PhD candidate at University of Texas at Dallas. His research interests include performance and memory optimization for embedded systems, and software/hardware codesign for parallel systems. He is a student member of the IEEE.



Qingfeng Zhuge received the PhD degree from the Department of Computer Science at the University of Texas at Dallas. She received the BS and MS degrees in electronics engineering from Fudan University, Shanghai, China. Her research interests include embedded systems, real-time systems, parallel architectures, optimization algorithms, high-level synthesis, compilers, and scheduling. She is a student member of the IEEE.

Meikang Qiu received the BE and ME degrees from Shanghai Jiao Tong University, Shanghai, People's Republic of China, in 1992 and 1998, respectively; he received the MS degree from the Department of Computer Science, University of Texas at Dallas, Richardson, Texas, in 2003. Now, he is a PhD candidate in the Department of Computer Science, University of Texas at Dallas. From August 1992 to August 2001, he was with the Department of Automation Control and Simulation at the Chinese Helicopter R&D Institute, IBM HSPC, and Shenzhen Quality and Technology Supervision Bureau. His research interests include embedded systems, information security, and mobile interface and intelligence. He is a student member of the IEEE.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, China, in 1997 and 2000 respectively, and the PhD degree from the University of Texas at Dallas in 2003 in computer science. Now, he is an assistant professor in the Department of Computing at the Hong Kong Polytechnic University. His research interests include computer networks and routing protocols, peer-to-peer communications, Internet security with a focus on denial of service (DoS) defense, embedded system design, mobile ad hoc networks, and wireless communication systems. He is a member of the IEEE.

Edwin H.-M. Sha received the BSE degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986; he received the MA and PhD degrees from the Department of Computer Science, Princeton University, Princeton, Nre Jersey, 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, Indiana. He served as 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 200 research papers in refereed conferences and journals. He has served as an editor for many journals and as a program committee member and chair for numerous international conferences. He received the Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, and US National Science Foundation CAREER Award. He is a senior member of the IEEE.