

# Defending Embedded Systems Against Buffer Overflow via Hardware/Software \*

Zili Shao, Qingfeng Zhuge, Yi He, Edwin H.-M. Sha  
Department of Computer Science  
University of Texas at Dallas  
Richardson, Texas 75083, USA  
{zxs015000, qfzhuge, yxh011010, edsha}@utdallas.edu

## Abstract

*Buffer overflow attacks have been causing serious security problems for decades. With more embedded systems networked, it becomes an important research problem to defend embedded systems against buffer overflow attacks. In this paper, we propose the Hardware/Software Address Protection (HSAP) technique to solve this problem. We first classify buffer overflow attacks into two categories (stack smashing attacks and function pointer attacks) and then provide two corresponding defending strategies. In our technique, hardware boundary check method and function pointer XOR method are used to protect a system against stack smashing attacks and function pointer attacks, respectively.*

*Although the focus of the HSAP technique is on embedded systems because of the availability of hardware support, we show that the HSAP technique can be applied to any type of processors to defend against buffer overflow attacks. We use four classes of processors to illustrate that the applicability of our technique is independent of architectures. We experiment with our HSAP technique in ARM Evaluator-7T simulation development environments. The results show that our HSAP technique can defend a system against more types of buffer overflow attacks with little overhead than the previous work.*

## 1 Introduction

It is known that buffer overflow attacks have been causing serious security problems for decades and at least 50% of today's widely exploited vulnerabilities are caused by buffer overflow and the ratio is increasing over time. One of the early famous examples is the *Internet worm* in 1988 that made use of buffer overflow vulnerabilities in *fingerd* and infected thousands of computers [1–3]. Recent examples include the infamous *Code Red*, *Code Red II* and their variations which exploited known buffer overflow vulner-

abilities in the Microsoft Index Service DLL. Another example, *Sapphire* or *SQL Slammer*, which occurred in the end of January of 2003, seriously slowed down the network by generating massive amount of traffic. This worm especially caused serious problems for the networks in Asia. It exploited the buffer overflow vulnerabilities in MS SQL server so its own hostile code can be executed.

Buffer overflow attacks cause serious damages to general purpose systems as well as to special purpose embedded systems. Because of the growing deployment of networked embedded systems, security becomes one of the most significant issues for embedded systems. Many special purpose systems are used in military and other critical 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 in such facilities could cause dramatic damages.

The design of embedded systems is not easy due to the strict requirements in latency, throughput, power consumption, area, cost, etc. With the increasing complexity of embedded applications, it becomes more attractive and necessary to design an embedded system by integrating as many off-the-shelf components as possible. A serious problem arises: how to check whether these components have buffer overflow vulnerabilities and how to protect an embedded system application from buffer overflow attacks? Note that the source code of most components is not available to system integrations, so it is hard to use a pure language-based approach to solve this problem. A hardware design is necessary if the software method is unable to achieve some of the requirements. Since hardware/software co-design is a common practice in embedded system designs such as using FPGA design or adding new instructions, we believe that an effective solution to protect embedded systems against buffer overflow attacks should lie on the combination of hardware and compiler. This paper will propose a method that uses both hardware and software to defend a system against buffer overflow attacks even without the knowledge of the source code.

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

This paper presents an effective approach with little overhead, called *HSAP: Hardware/Software Address Protection*, which can avoid stack overflows and minimize the damage of any other overflows such as heap overflow or BSS overflow even in the presence of insecure third-party components without the knowledge of the source code. We classify overflow-based attacks into two categories: *stack smashing attacks* and *function pointer attacks*. HSAP approaches each of them with different mechanisms. For *stack smashing attacks*, the most common attacks, our method can completely protect a system against these attacks. For *function pointer attacks*, our method will make it extremely hard for a hacker to change a function pointer leading to the hostile code. More specifically, HSAP achieves the following properties:

- For stack smashing attacks, the most common attacking method, it completely protects the return address, frame pointers, and arguments from being overwritten in the stack. Therefore, the system is shielded from this type of attacks.
- For function pointers such as shared function pointers in GOT (Global Offset Table) or local function pointers in heap or BSS (Block Storage Segment), HSAP makes it extremely hard for a hacker to change any function pointer to point the new address to the beginning of the inserted hostile code. This is enforced by the hardware instructions to guarantee that jump addresses are encrypted so any third-party software components must follow the secure instructions in order to make the code runnable.

Although the focus of the *HSAP* technique is on embedded systems because of the availability of hardware support in embedded system designs, we show that the *HSAP* technique is general enough to be applied to various types of processors. Based on the different stack structures, we classify processors into 4 classes and show the *HSAP* technique can be applied to all these four classes of processors and is independent of architectures. We experiment with our HSAP technique in ARM Evaluator-7T simulation development environments. Our experiments results show that *HSAP* can defend a system against more types of buffer overflow attacks with little overhead compared with the previous work.

The remainder of this paper is organized as follows. Section 2 gives a brief discussion of related work. In section 3, examples are given to show the basic buffer overflow attack methods that provide the necessary background for understanding our approach. Our Hardware/Software defending approach, HSAP, is presented in Section 4. Section 5 discusses the implementation of our HSAP approach on various processors. The performance comparison and experiments are presented in Section 6. Section 7 concludes this paper.

## 2 Background and Related Work

Buffer overflow vulnerabilities appear where an application reads external information into a buffer using vulnerable library function. The possible places that have overflow vulnerabilities include stack, heap, BSS (Block Storage Segment), or any other places that store variables. Some typical examples about basic buffer overflow attack methods are given in Section 3 based on the stack structure of Intel-like processors.

The most common attacks target at *stack overflows* because it is not hard to prepare a well-crafted buffer such that the return address is changed and set to the beginning of malicious code that was inserted in the buffer. Such attacks are often named *stack smashing attacks*. The basic knowledge of stack smashing attacks is introduced in [4,5].

The most common strategy to exploit stack overflows is as follows:

1. Find a program with stack-overflow vulnerabilities and prepare a buffer that can overflow its stack frame so that the return address is changed.
2. Send this buffer as the input to the target program.
3. Make the new return address jump to the inserted hostile code that was originally copied from the buffer.

An attack based on heap or BSS uses a similar idea. For example, it can change a function pointer in heap or BSS to point to the inserted hostile code. The heap/BSS-based overflows are becoming common today. The detailed descriptions of heap/BSS-based smashing attacks can be found in [6–8]. A new class of vulnerabilities, “format string bugs”, was disclosed in 2000. The detailed descriptions of the exploitation of printf vulnerabilities can be found in [9].

A common approach to solve 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 every off-the-shelf software is immune to buffer overflows. The use of the safe programming languages is effective to defend against buffer overflow attacks. But for embedded system applications, most software is still written in “unsafe” languages such as C or assembly. It is also impractical to require all software components to use the same language or to be compiled by the same compiler. Therefore, it is very hard for a system integrator to check buffer overflow vulnerabilities of a component based on a pure language-based approach.

The static checking method uses the strategy to detect the vulnerabilities by analyzing the code [10–13] by software tools. While such tools can greatly help programmers find the vulnerabilities of their code, the protection provided may be incomplete and imprecise, as only known vulnerabilities can be detected and the general buffer overflow detection problem is undecidable.

The dynamic checking method detects buffer overflow vulnerabilities during the program execution. Basically,

two different strategies, runtime boundary checking and program testing, are used in this method. Runtime boundary checking strategy [14–17] adds instructions to check array bounds and performs pointer checking in run time. While using this strategy may completely protect a system against buffer overflow attacks, a big performance overhead may occur especially for array and pointer intensive applications. Program testing strategy [18–21] 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 test data to cause overflows.

It is very common that the hostile code is inserted into the stack and then executed. Based on this, a strategy that can make the stack non-executable has been used to implement a Linux kernel patch that removes the stack execution permission [22]. However, this technique can not defeat a type of attacks using the return-into-libc technique, in which the vulnerable function can return into a memory area occupied by a dynamic library [23, 24]. Considering the problems caused by a non-executable stack (for example, function trampolines for nested functions need executable stacks), this strategy is not very applicable.

A method that intercepts vulnerable functions and forces verification of critical elements of stacks is proposed in [25]. Libsafe and Libverification are implemented as dynamically loadable libraries. The advantage of this method is that it doesn't need source code. However, it can not defend a system against heap/BSS-based smashing attacks [6–8].

Another promising approach uses compilers to automatically add extra instructions to guard stack at running time. The research efforts to guard the stack based on this approach include StackShield, StackGuard, IBM SSP, StackGhost, etc. [26–29].

- StackShield [27] is a modification of gcc. It protects a return address by storing it in a separate stack. It can only protect a system against a particular type of stack smashing attacks that need to overwrite return addresses.
- StackGuard, also a modification of gcc, is implemented based on the idea using a canary to guard return addresses [26]. StackGuard changes the prologue and epilogue of a function call. It pushes a canary right above a return address in the stack in the prologue and checks whether the canary has been changed in the epilogue. StackGuard is a reasonable way of defending a system against buffer overflow attacks that overwrite the return address.
- Microsoft's /GS protection [30] uses the similar idea that StackGuard uses, and it further provides the protection for the frame pointer. In Microsoft's /GS protection, the random canary is put between the frame pointer and local variables.

- IBM SSP [28] uses the similar strategy to protect the frame pointer and return address (random canary). Furthermore, it provides protections for local variables and function's arguments. In IBM SSP, local variables are reordered in such a way that pointers are placed before a possibly attacked buffer to avoid the corruption of pointers. And pointers in function arguments are copied to an area preceding a local variable buffer [28].
- Under Sun Microsystem's Sparc processor architecture, some techniques to protect the return address by modifying the kernel are proposed in [29]. A tool called StackGhost is implemented to transparently and automatically guard applications by XOR-ing return address in the kernel. StackGhost is based on the specific hardware platform and needs to revise the kernel. So it can not be applied to solve the general stack smashing attack problems.

StackShield and StackGuard can defend against stack smashing attacks that overwrite the return address. However, they have four flaws:

1. They don't protect function arguments and local variables, which are used to exploit in [31, 32].
2. They don't protect frame pointers that are used to exploit in [32, 33].
3. They detect an attacks after a function finishes, which gives hackers a code window to play with and exploit [6–8, 31, 32].
4. They need source code so they may not be applied to protect third party software components that only provide the executable code.

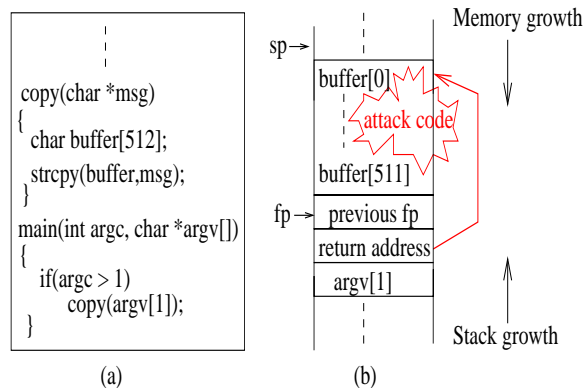
Microsoft's /GS provides the protection for frame pointers so it can avoid the vulnerabilities in flaw 2. IBM SSP can avoid flaw 1 and 2 subject to some limitation for certain source programs. It can not avoid the vulnerabilities in flaw 3 if an attacker combines heap/BSS-based smashing attacks [6–8] with the stack smashing attacks [31, 32]. Considering flaw 4, none of these techniques can be applied to special purpose systems such as aircraft control systems which use lots of third party software components.

In [34], PointGuard is proposed to protect pointers from buffer overflow attacks by encrypting pointer values while they are in memory and decrypting them before dereferencing. We use the similar idea to protect function pointers. While PointGuard can protect a system from the vulnerabilities in flaw 1-3, it needs source code and requires programmers to manually add PointGuard protection, which limit its applicability somehow for embedded system protection.

### 3 Buffer Overflow Attacks

In this section, the examples for basic buffer overflow attack methods are shown to provide the background for

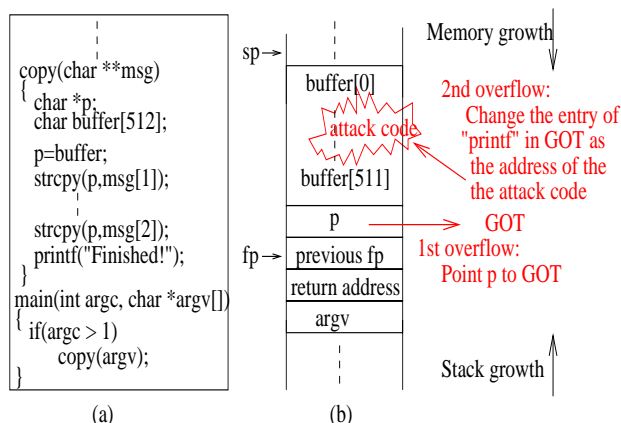
understanding why our approach is necessary and how it works. A common stack smashing attack example is shown first. Then we give two advanced stack smashing attack examples. Finally, a BSS overflow attack example is given. The stack structure of Intel x86 processors (see Section 5 for information on other processor types) is used in these examples because it is easy to explain and understand. More issues related to architectures are discussed in Section 5.



**Figure 1. (a) Vulnerable program 1. (b) The stack structure and attack.**

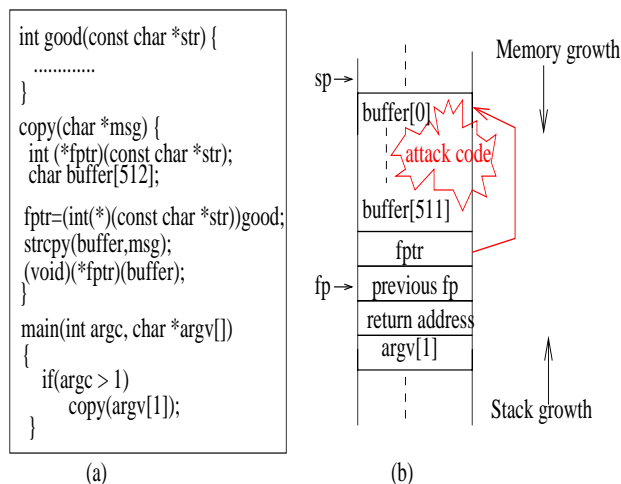
Vulnerable program 1 is shown in Figure 1(a). Figure 1(b) shows a typical stack structure after function `copy()` is called, where arguments, return address, previous frame pointer, and local variables are pushed into the stack one by one. The arrows in Figure 1(b) show the growth directions of stack 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 Figure 1(b). Then the attack code will be executed after the program returns from `copy()` to `main()`. Almost all techniques introduced in Section 2 can defend against this kind of attacks. 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()`, the similar attack can be activated when returning from `main()`. It is used in [32, 33] to defeat the protections of StackShield and StackGuard.

Even if the whole stack frame is protected, we still can not defend against the stack smashing attack shown in Figure 2. The vulnerable function `copy()` in Figure 2(b) has one local pointer `p` and calls `strcpy()` twice. The location of `p` (Figure 2(b)) 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 overflowed to point to the entry of `printf` in the shared function pointer table GOT; 2) The address of the attack code is copied to the entry of `printf` in GOT by the second `strcpy()`; 3) The attack code is activated when the program



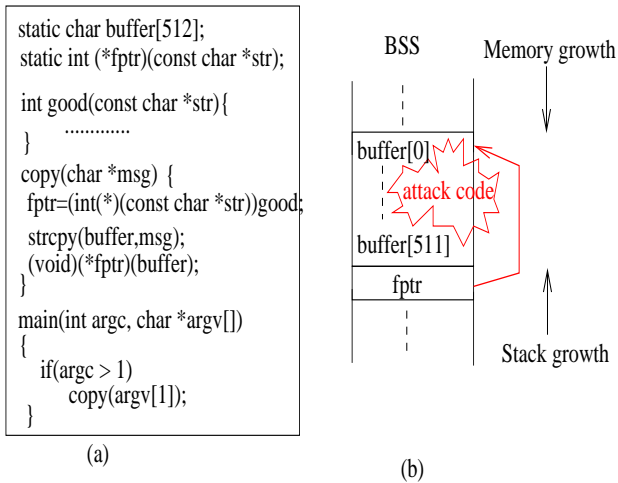
**Figure 2. (a) Vulnerable program 2. (b) The stack structure and attack.**

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 attacks.



**Figure 3. (a) Vulnerable program 3. (b) The stack structure and attack.**

Our third example (Figure 3) shows that the local function pointer can be exploited in the stack. In this vulnerable program, `copy()` (Figure 3(a)), `strcpy()` is called first and then the function pointed by function pointer `fptr` is executed. In the stack (Figure 3(b)), the location of `fptr` is below the location of `buffer[]`. Thus, `fptr` can be overflowed and pointed to the attack code by `strcpy()`. Then, the execution of `(void)(*fptr)(buffer)` will activate the attack code. Figure 4 shows a similar example to exploit the local function pointer in BSS. These examples show that local function pointers need to be protected to defend against this kind of attacks.



**Figure 4. (a) Vulnerable program 4. (b) The BSS structure and attack.**

## 4 HSAP: Hardware/Software Address Protection

In this section, we first classify buffer overflow attacks into two categories. Then two corresponding defending strategies are proposed. Finally, security analysis is performed.

### 4.1 The Categories of Buffer Overflow Attacks

It is extremely hard to design a pure hardware scheme that protects all variables in stack, heap, BSS, or any data segments, from being overwritten by buffer overflows. This is because, from the hardware point of view, 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 little overhead so that a secure real-time embedded system can be built.

We can classify overflow-based attacks into two categories:

1. **Stack smashing attacks.** This attack either overwrites a return addresses as shown in Figure 1 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.
2. **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 Figure 3-4 or indirectly by using a similar method as shown in Figure 2. 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 Figure 2.

*Function pointer attacks* are not as easy to exploit by a hacker as stack smashing attacks.

Our goal is to make sure that it is extremely hard for a program to run an inserted hostile code by overflowing any variables in stack, heap, or BSS. Once this goal is achieved, the hostile program in a buffer can not be run by buffer overflows. Moreover, 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 *local function pointer attacks*, our goal is to make a hacker extremely hard to change a function pointer leading to his hostile code.

### 4.2 The HSAP Technique

#### 4.2.1 Basic Concepts

Before describing the details of our scheme, we need the following definition.

**Definition 4.1.** A **jump pointer** is a location that stores an address to a code segment. A program can jump to the address stored in a jump pointer.

A possible jump pointer can be a location that stores a return address in a stack frame, or a location that stores a function pointer, or an entry in a shared function pointer table such as GOT. The goal of our technique is to make sure no jump pointers can be changed to point to the inserted hostile code.

The existing protection schemes using special compiler features such as StackGuard, StackShield, IBM SSP, Microsoft /GS, etc., do not provide complete protection for jump pointers. None of these techniques can protect jump pointers in the heap or BSS. As shown in Figure 4, a hacker can deploy a *local function pointer attack* by changing a function pointer to point to the attack code. Although PointGuard can protect such jump pointers, it requires source code, which may not be provided for most third party software components.

The technique we propose, called *HSAP: Hardware/Software Address Protection*, provides the following protections:

1. Protect return addresses from being overflowed.
2. Protect *frame pointers* from being overflowed.
3. Protect any other jump pointers (function pointers) from being exploited.

Since one of the major applications of HSAP is the design of a secure and real-time embedded system, in addition to the concern of security, the performance is another serious requirement. It is not tolerable to have significant performance loss using software approaches. We must study special hardware support to minimize performance loss and prove that the security requirements are met. We also need to create an effective check and protection mechanism to deal with insecure third party components without knowing the source code.

HSAP requires two components: **stack smashing protection** and **function pointer XOR**. We will explain why HSAP needs both of them later.

#### 4.2.2 Component 1: Stack Smashing Protection

The protection scheme is to perform *hardware boundary check* using the current value of the frame pointer. The basic approach is:

1. An “address check” phase is added before the “write” phase in the pipeline.
2. The write operation is denied when the target’s address is equal to or bigger than the value of frame pointer.

The implementation of this hardware boundary check on different processors may be different because of different stack structures. The implementation details can be found in Section 5.

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 overflowing buffer. Therefore, we can defend a system against *stack smashing attacks* completely. Second, boundary checking is implemented by a pipeline phase, so there is little performance overhead ( $\approx 0$ ). Finally, the source code and the extra protection code are not needed.

Our *stack smashing protection* technique requires that third party software developers put variables into data/BSS/heap section (define as global variables, static variables, or dynamic memory allocation) rather than stack section (define as local variables) if these variables needs to be changed in further function calls. To release this requirement, new stack smashing protection techniques have been developed in [35].

#### 4.2.3 Component 2: Function Pointer XOR

To protect function pointers from being exploited, we use the following approach:

- Randomly assign a *key* to each process when it is generated. Keep the *key* in a special register R.
- Add a new jump instruction, SJMP, to an instruction set whose operations are: first XOR the input address with R (the *key*), and then jump to this XORed address. For example, “SJMP A” will include two operations: (1)  $A' \leftarrow A \text{ xor } R$ ; (2)  $\text{JMP } A'$ , The XOR operation is done automatically by the hardware when the jump instruction is executed. We propose to enhance the common instruction set with additional instructions for security purpose. Then any third party must use these secure instructions to write their code to comply with security requirements and this enforcement can be easily checked with their binary code.

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”.

For the third party, these requirements can be easily achieved. For example, they can revise the linker and the dynamic linker to process local function pointers and shared function pointers, respectively. As a system integrator, we can easily check whether SJMP has been used to secure all function pointer calls with their binary code. Therefore, the source code is not needed for security check.

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 jumps to is the XORed address with the key. Since the *key* is randomly generated for each process, it is extremely hard for a hacker to guess the *key*. The *key* is stored in a special register, therefore, the key value cannot be overwritten by buffer overflow attacks.

#### 4.2.4 The Necessity of Two Components

Both of these two components are necessary. *Function pointer XOR* is required to secure *local function pointers* and *shared function pointers*. Though we can use the similar idea to protect the return addresses, it cannot protect frame pointers and arguments in a stack. So we need stack smashing protection. Using these two components together, we can achieve a sound solution for buffer overflow attacks with little overhead. Both components can be efficiently implemented on various processors, which is discussed and analyzed in Section 5.

### 4.3 Security Analysis

The HSAP 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 a heap or BSS. Then an attack through a file name like in [1–3] 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 “jump pointer” to the external file system. This requires software approach to protect the integrity of any file structures. 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.

## 5 Application of HSAP to Various Processors

In this section, we show that the HSAP technique can be applied to any type of processors to defend against buffer overflow attacks. We use the four classes of processors to illustrate that the applicability of our technique is independent of architectures.

### 5.1 Processor Class 1 (Intel-x86-like processors)

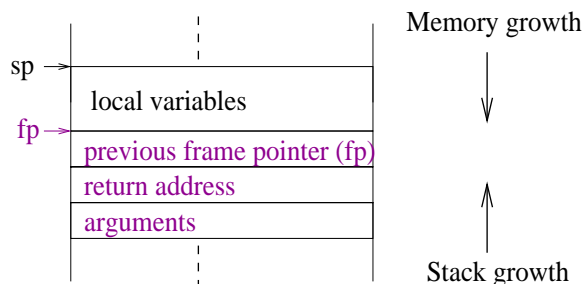


Figure 5. A typical stack structure for class 1 processors.

For class 1 processors, a typical stack structure after a function is called is shown in Figure 5. When a function is called in this class of processors, arguments, return address, and frame pointer (fp) are pushed into the stack sequentially. Then the value of fp is changed to point to the address that stores the previous frame pointer in the stack. Next, the stack pointer (sp) is decreased to leave space for local variables. When returning from the function, a “ret” instruction will be called. In this “ret” instruction: 1) The stack pointer (sp) is changed to point to the address that stores the previous frame pointer based on the value of fp; 2) The previous frame pointer is popped to fp; 3) The return address is popped to Program Counter. Then the execution is returned to the caller in the next cycle.

Since arguments, return address and the previous frame pointer all are stored below fp, they are protected if a write

operation is denied when its address is equal to or bigger than fp. So we can directly implement *stack smashing protection* of HSAP technique on this class of processors by checking the address of a write based on fp in the pipeline. For the second component, *Function Pointer XOR*, the required secure instructions can be added into a secure instruction extension and easily implemented because they only need very simple hardware.

### 5.2 Processor Class 2 (Sun-Sparc-like processors)

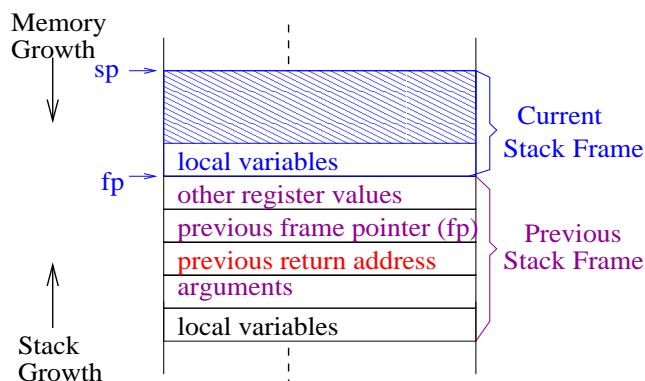


Figure 6. A typical stack structure for class 2 processors.

The stack structure of class 2 processors (Figure 6) is similar to that of class 1 processors. However, there are two main differences between them. First, in class 2 processors, the return address of the current function is stored in a special register instead of in the stack in class 1 processors. For example, it is stored in register i7 in Sun Sparc processors. There is still one return address stored in the stack (*previous return address* in Figure 6) when a function is called, but it is the return address of a caller. Using the C program in Figure 1(a) as an example, in class 2 processors, after function copy() in main() is called, the *previous return address* in the stack in Figure 6 is the return address of main(), and the returning address of copy() is stored into a register. Since the return address is stored in a register in class 2 processors, a buffer overflow attack can only change the value of *previous return address* in an attack. Therefore, it makes attacks a little harder. But an attack can still be activated after returning from a caller. Second, the space in the stack is left not only for local variables but also for the calling information. If there are no function calls in a function, then the calling information is empty; otherwise, the calling information such as arguments, previous return address, previous frame pointer, etc., will be put into this area ( the shadow area in Figure 6).

In spite of these differences, the application of the HSAP technique on class 2 processors is the same as that on class 1 processors because arguments, previous return address and the previous frame pointer all are stored below fp.

### 5.3 Processor Class 3 (ARM-like processors)

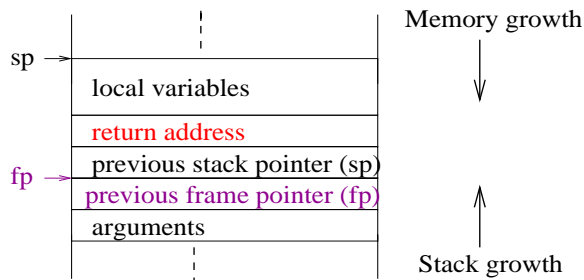


Figure 7. A typical stack structure for class 3 processors.

In class 3 processors, the sequence of the parameters in the stack (Figure 7) is different from that in class 1 processors when a function is called. Note that the addresses that store return address and the previous stack pointer are less than the address that fp points to. Thus, to protect these values from the buffer overflow attacks, we need to check the address that decreases a constant from the value of fp in the hardware boundary check. This constant is decided by the real hardware. For example, in ARM 32-bit processors, this constant should be 8.

### 5.4 Processor Class 4 (TI-5x-like processors)

The stack structure for class 4 processors is shown in Figure 8 when a function is called. In this class of processors, there is no frame pointer. Since sp will be changed in the function later, we can not use it as the base address in hardware boundary check. To make our HSAP work, we need to add a register to record the address that stores return address when a function is called. This can be simply implemented as follows: when a “call” instruction is executed (to call a function), extra functions are added into this instruction so that return address is written to our register. Then in the address check phase in the pipeline, we will compare the address of a write with the value of this register. Using this implementation, we don’t need to change the source code.

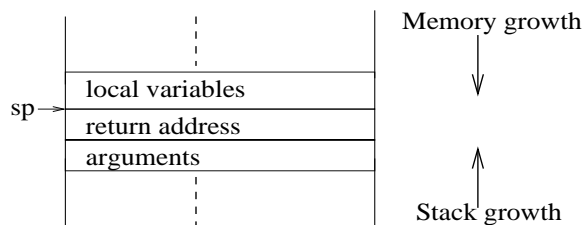


Figure 8. A typical stack structure for class 4 processors.

## 6 Comparison and Experiments

In this section, we first compare overhead caused by our HSAP technique with that caused by PointGuard and

StackGuard, respectively. Then we experiment with our HSAP technique in ARM Evaluator-7T simulation development environments and summarize the protection effectiveness.

### 6.1 The Overhead Comparison

#### 6.1.1 HSAP vs. StackGuard

Stack Smashing Attacks		
	StackGuard	HSAP
Overhead	7 additional instructions for each call	$\approx 0$
Support	Compiler	Hardware

Table 1. The comparison of HSAP and StackGuard.

To protect against *stack smashing attacks*, StackGuard needs seven additional instructions for each function call. In our HSAP technique, the stack smashing protection is implemented by adding a pipeline stage to do hardware boundary check. Therefore, the overhead introduced approximates to 0. While HSAP needs the hardware support to achieve this protection, StackGuard needs the compiler support. Table 1 summarizes the comparison results.

#### 6.1.2 HSAP vs. PointGuard

PointGuard protects all pointers by encrypting their values and decrypting them before dereferencing (an option is given to manually select whether or not protect a pointer). Our technique uses the similar idea to protects *function pointers* only. Since *function pointers* are basic pointers that must be protected in PointGuard, the overhead introduced by HSAP is less than or equal to that introduced by PointGuard. Still HSAP provides the same level of security as PointGuard does in term of function pointer protection because a corrupted pointer finally needs to change a jump pointer to activate a buffer overflow attack.

### 6.2 Experiments

We experiment with our HSAP technique in ARM Evaluator-7T simulation development environments. The simulation development environments are built up on a Linux PC (Red Hat Linux 7.3) based on the methods introduced in [36]. The environments include a ARM-elf GNU cross compiler, a GNU debugger, and a C runtime library. The GNU debugger is used as a ARM instruction set simulator. Based on [6–8, 31–33], we implement various C vulnerable programs and corresponding attack programs. They cover all known types of attacks on different places including stack, heap and BSS. We compare our HSAP technique with StackGuard, StackShield, IBM SSP, and LibSafe introduced in Section 2. In the experiments, we first obtain the assembly code using ARM-elf

Methods	Source Code Needed for S.C.	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
HSAP	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
StackGuard	Yes	No	No	Yes	No	No	No	No	No	No
StackShield	Yes	No	No	Yes	No	No	No	No	No	No
IBM SSP	Yes	Yes	Yes/No	Yes	No	No	No	No	No	No
LibSafe	Yes	Yes	Yes	Yes	No	No	No	No	No	No
PointGuard	Yes	No	Yes/No	No	Yes	Yes	Yes	Yes	Yes	Yes

**Table 2. Security effectiveness comparison.**

cross compiler, and then secure the assembly code based on different methods. Finally, the secure assembly code is executed on the GNU debugger’s ARM instruction set simulator.

Table 2 summarizes the test results. 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 “attack” ; “Yes” denotes that the method succeeds; “Yes/No” denotes that the method either fails or succeeds for different programs. Column “Source Code Needed for S.C.” lists whether the source code of a component is needed when checking whether the component has been protected by this method.

The results in Table 2 show that our HSAP technique can defend more types of buffer overflow attacks. All methods except HSAP and PointGuard can not protect against the shared function pointer attacks and local function pointer attacks. To protect against the stack smashing attacks, StackGuard and StackShield can only protect against the attacks that overflow return address. Both of them fail to protect against the stack smashing attacks by changing fp or arguments. IBM SSP can basically protect against the stack smashing attacks. But it fails in some cases when attacks are through arguments. For example, it fails when the program has a structure that has a pointer and a character array. To check whether a software component has been protected, HSAP doesn’t need the source code while all other methods do.

## 7 Conclusion

In this paper, we proposed the Hardware/Software Address Protection (HSAP) technique to defend embedded systems against buffer overflow attacks. We first classified buffer overflow attacks into two categories and then provided two defending components correspondingly. We showed that our HSAP technique can be applied to various processors. We experimented our HSAP technique in ARM on ARM Evaluator-7T simulation development environments. The experimental results show that our HSAP technique can defend more types of buffer overflow attacks with little overhead compared with the previous work.

## 8 Acknowledgments

We would like to thank Jeremy Epstein for his careful review and insightful comments.

## References

- [1] E. H. Spafford, “The internet worm program: an analysis,” Tech. Rep. TR823, Purdue University, 1988.
- [2] Mark M. Eichin and Jon A. Rochlis, “With microscope and tweezers: An analysis of the internet virus of november 1988,” in *Proc. of the IEEE Computer Society Symposium on Security and Privacy*, 1989.
- [3] D. Seeley, “A tour of the worm,” in *Proc. of the USENIX Annual Technical Conference*, Winter 1989.
- [4] Aleph1, “Smashing the stack for fun and profit,” *Phrack*, 7(49), Nov. 1996.
- [5] pr1, *Exploiting SPARC buffer overflow vulnerabilities*, World Wide Web, <http://www.u-nf.com/papers/UNF-Sparc-Overflow.html>, 2003.
- [6] Matt Conover and w00w00 Security Team, *w00w00 on Heap Overflow*, World Wide Web, <http://www.w00w00.org/articles.html>, January 1999.
- [7] Michel Kaempf, “Vudo - an object superstitiously believed to embody magical powers,” *Phrack*, 8(57), August 2001.
- [8] anonymous author, “Once upon a free(),” *Phrack*, 9(57), August 2001.
- [9] scut / team teso, *Exploiting Format String Vulnerabilities*, World Wide Web, <http://www.team-teso.net/articles/formatstring>, September 2001.
- [10] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, “A first step towards automated detection of buffer overrun vulnerabilities,” in *Network and Distributed System Security Symposium*, San Diego, CA, February 2000, pp. 3–17.

- [11] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw, "Token-based scanning of source code for security problems," *ACM Transactions on Information and System Security*, vol. 5, no. 3, pp. 238–261, 2002.
- [12] David Larochelle and David Evans, "Statically detecting likely buffer overflow vulnerabilities," in *Proc. of the USENIX Security Symposium*, August 2001.
- [13] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *the 10th USENIX Security Symposium*, August 2001, pp. 201–206.
- [14] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Winter USENIX Conference*, 1992, pp. 125–136.
- [15] 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*, June 1994, pp. 290–301.
- [16] Richard W. M. Jones and Paul H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *the third International Workshop on Automated and Algorithmic Debugging*, 1997, pp. 13–26.
- [17] Kyung suk Lhee and Steve J. Chapin, "Type-assisted dynamic buffer overflow detection," in *Proc. of the USENIX Security Symposium*, August 2002, pp. 29–40.
- [18] 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.
- [19] G. Fink, M. Helmke, M. Bishop, and K. Levitt, "An interface language between specifications and testing," Tech. Rep. CSE-9515, University of California, Davis, 1995.
- [20] G. Fink and M. Bishop, "Property-based testing: A new approach to testing for assurance," *ACM SIGSOFT Software Engineering Notes*, p. 22(4), July 1997.
- [21] Eric Haugh and Matt Bishop, "Testing c programs for buffer overflow vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2003.
- [22] Solar Designer, *Kernel patches from the openwall project*, World Wide Web, <http://www.openwall.com/linux>, 2002.
- [23] Rafel Wojtczuk, *Defeating solar designer non-executable stack patch*, World Wide Web, <http://www.insecure.org/spl0its/non-executable.stack.problems.html>, 1998.
- [24] Nergal, "The advanced return-to-lib(c) exploits: Pax case study," *Phrack*, 4(58), December 2001.
- [25] A. Baratloo, T. Tsai, and N. Singh, "Transparent runtime defense against stack smashing attacks," in *Proc. of the USENIX Annual Technical Conference*, June 2000.
- [26] 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.
- [27] Vindicator, *Stack Shield: A stack smashing technique protection tool for Linux*, World Wide Web, <http://www.angelfire.com/sk/stackshield/info.html>, 2000.
- [28] Hiroaki Etoh and Kunikazu Yoda, *Protecting from stack-smashing attacks*, World Wide Web, <http://ww.trl.ibm.com/projects/security/ssp/main.html>, May 2002.
- [29] Mike Frantzen and Mike Shuey, "Stackghost: hardware facilitated stack protection," in *Proc. of the USENIX Security Symposium*, August 2001.
- [30] Brandon Bray, *Compiler security checks in depth*, <http://go.microsoft.com/fwlink/?LinkId=7260>, Feb. 2002.
- [31] Bulba and Kil3r, "Bypassing stackguard and stackshield," *Phrack*, 5(56), May 2000.
- [32] 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.
- [33] klog, "The frame pointer overwrite," *Phrack*, 8(55), September 1999.
- [34] 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.
- [35] Zili Shao and Edwin Sha, "Efficient security check and protection for embedded system against buffer overflow attacks," Submitted.
- [36] Willam Gatliff, "Getting started with gnu: a tutorial introduction using the arm evaluator-7t," *Circuit Cellular Ink*, Dec. 2001.