SmashClean: A Hardware level mitigation to stack smashing attacks in OpenRISC

Manaar Alam, Debapriya Basu Roy, Sarani Bhattacharya, Vidya Govindan,
Rajat Subhra Chakraborty and Debdeep Mukhopadhyay
Secured Embedded Architecture Laboratory, IIT Kharagpur, India,
Email: {alam.manaar, vidya.mazhur}@gmail.com, {deb.basu.roy, sarani.bhattacharya, rschakraborty, debdeep}@cse.iitkgp.ernet.in

***Abstract*—Buffer overflow and stack smashing have been one of the most popular software based vulnerabilities in literature. There have been multiple works which have used these vulnerabilities to induce powerful attacks to trigger malicious code snippets or to achieve privilege escalation. In this work, we attempt to implement hardware level security enforcement to mitigate such attacks on OpenRISC architecture. We have analyzed the given exploits [5] in detail and have identified two major vulnerabilities in the exploit codes: memory corruption by non-secure `memcpy()` and return address modification by buffer overflow. We have individually addressed each of these exploits and have proposed a combination of compiler and hardware level modification to prevent them. The advantage of having hardware level protection against these attacks provides reliable security against the popular software level countermeasures.**

I. Introduction

An embedded system handling sensitive information can be subjected to various security threats via software and hardware based vulnerabilities. In general, light-weight embedded system applications use C and C++ language framework due to better performance compared to other languages. However, low-level languages like C and C++ does not provide secure memory management methodologies. Hence they can be subjected to memory corruption through buffer overflow. This buffer overflow in the memory occurs due to the absence of any bound check during memory update. Hence, an intelligent adversary can easily overwrite the memory with desirable parameters to trigger malicious code execution which can compromise the security of the entire system. Software level countermeasures which claim to prevent memory corruption can be itself bypassed by advance malwares. Hence, implementing hardware level security policies on the processor itself is more secure option compared to software based countermeasures.

Particularly in this case we summarize some interesting hardware level mitigation techniques in literature. In [6], Nagarakatte et. al., proposed a hardware-based approach, Watchdog, to ensure safe and secure manual memory management. This scheme generates a unique identifier for each memory allocation and checks whether the identifier is valid on every memory access to detect abnormalities while memory access.

A hardware-based protection of the function return address stored on the program stack is presented by Ozdoganoglu et. al. in [8]. This method, with negligible performance overhead, adds a small hardware stack to the processor which is inaccessible to the user process. With each function call the return address is pushed onto the process stack and simultaneously on the hardware stack. Each return statement pops the return addresses from top of both the stacks and compares them. An exception is raised with a mismatch. On the other hand, Song et. al., proposed a new fine-grained data isolation mechanism for protection against the memory corruption vulnerabilities [10]. Their hardware-assisted data-flow isolation (HDFI) mechanism ensures isolation at the machine word granularity with an additional extended tag for each memory unit. The target architecture for each of these existing works are different and thus are not directly applicable in our implementation.

We aim to present an approach which combines the compiler as well as hardware modification to prevent all forms of memory corruption and buffer overflow attacks on OpenRISC architecture. In this work our objective is to introduce new instructions by modifying the hardware which will be used by the compiler to detect and prevent memory corruption via buffer overflow. We have identified two genre of serious threats in the given exploits. `memcpy()` being one of the common function for updating memory does not impose any bound-checking during memory update and thus is vulnerable to buffer overflow attacks.

As protection mechanism, new customary instruction have been introduced in this work by modifying the hardware architecture of OpenRISC (or1k cappuchino). These new instructions keep track of the buffer size and ensures number of memory locations upgraded by `memcpy()` is less than or equal to buffer size. Additionally, return address modification can be prevented by storing the return addresses in the hardware stack. If the buffer overflow corrupts the return address of the memory, hardware stack ensures the correct control flow of the program is maintained.

The structure of the paper can be given as follows. In section II, the basic exploitation techniques for memory corruption and control-flow based attacks are briefly explained. In section III we discuss the proposed hardware mitigation techniques followed by the result and performance in secton IV. Finally, we conclude our work in section V.

II. Exploitation Methods

Normal execution flow of any system is often subjected to external attacks. These attacks aim to control the behavior of a system and arrive as a normal data over a regular communication channel. Once the data resides in program memory they trigger any pre-existing software flaws. The attacker exploits such flaws to subvert execution and gain control over software behavior. One popular example of these kind of attacks is - *buffer overflow exploit* [7], [9].

978-1-5090-2791-0/16/$31.00 ©2016 IEEE

Buffer overflow occurs when a program attempts to read or write beyond the end of a bounded array (also known as a buffer). Some well documented widely used string manipulation functions, such as `strcpy()`, `strcmp()`, `memcpy()`, are generally used with an array variable as their arguments. These functions are normally vulnerable if some form of bound checking is not used along with them. The content of the user input, which is beyond the bound of the buffer may interfere with some private data or address of a function. An attacker gains knowledge about the address of the malicious function beforehand due to the *Memory Disclosure Vulnerability*. The adversary takes advantage of this flaw to call a function that the application never intended to use.

Following example briefly explains the `buffer overflow` exploit [1] with insufficient bound checking. A program has two data variables which are adjacent to each other in the memory location. An 8-byte long `string` buffer `A` and a 2-byte long `integer B`.

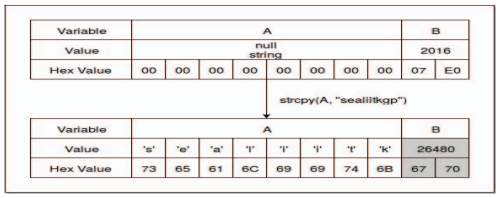


Fig. 1: Example of Buffer Overflow

As described in Fig. 1, `sealiitkgp` is a 11 characters long string including the terminator, but `A` can take only 8 bytes. Since there is no bound checking for the length of the string, `strcpy()` overwrites the value of `B`. The value of `B` is now unintentionally replaced by a number which is created from the part of the character string `A`. In this example, `g` followed by `p` produces `26480`.

When combined, these *memory corruption* and *control flow integrity* attacks make one of the most critical challenges in computer security. In this section we start with discussing some of the common yet challenging system exploits that we intend to mitigate in this literature.

A. *Return Address Modification by Stack Buffer Overflow*

In this traditional approach the attacker modifies the return address saved on the stack (the region of memory used for storing return address, parameters and local variables), to point to a known address of a malicious function that the attacker supplies as an argument [7].

The code snippet in Table I explains this kind of attack.

B. *Format String Vulnerability*

Format string vulnerabilities usually occur when a programmer wants to print out a user controlled function but does not sanitize the user input [11]. This allows the malicious attacker to inject their own format specifiers, which in turn allows the adversary to read and write arbitrary memory.

The following code snippet contains such an error.

```
int func(char* user) {
  printf(user);  //Vulnerability
}
```

```
int n;
printf("%12c%n", 'A', &n);
```

Since `printf` has a variable number of arguments, it must use the format string to determine the number of arguments. In the case above, the attacker can pass any string of his choice and fool the `printf` function from arbitrary write primitive into arbitrary code execution.

For example, `%n` is a format specifier that has the following effect. It stores the number of characters written so far into an integer indicated by the `int*` pointer argument. The variable `n` will contain `12` after the `printf` call in second column.

An adversary can utilize this exploit by passing any malicious format string to the program and write desired values into the private memory location.

C. *Function Pointer Modification*

These types of attack are especially useful when some form of mitigation technique is used to prevent the modification of the saved return address of the program being attacked [9]. This attack modifies a function pointer to point to attacker-supplied code. When a function pointer is used in the program to execute a function call, the attacker's code is executed instead of the originally intended code.

The code snippet in Table I explains these kinds of attack.

D. *Data Pointer Modification*

Insecure use of `memcpy()` function can overwrite a buffer and modify adjacent memory locations on the stack [5], [9]. These types of attacks occur when the modified memory locations correspond to data pointers or data variables. If such variables are assigned to dereference of a pointer (i.e., a statement of the form `*ptr=data`), it may be possible to modify any memory location in the scope of the program.

Code snippet in Table I can be taken as an example for this kind of attack.

III. Hardware Mitigation

In previous sections we have introduced the notions of software based buffer overflow attacks. We have also analyzed the given exploits in detail. In this section, we will introduce our mitigation techniques which are mainly hardware based countermeasure, requiring a compiler level modification for integration with operating system.

The given sample exploits target two different types of vulnerabilities: memory corruption and control flow corruption. In case of control flow corruption, the exploits modify the return address of the function to trigger a malicious activity (malicious shell code execution). On the other hand, memory corrupting exploits modify the local variables which in turn lead to malicious function execution and privilege escalation. The buffer overflow happens due to non-secure `memcpy()` function which updates the buffer without any bound check. Memory corrupting exploits can be protected by securing the `memcpy()` function. Whereas, control flow corrupting exploits can be mitigated by protecting the function return address. In the subsequent section, we will focus on these two aspects and will propose two mitigation technique for these two type of vulnerabilities.

A. *Enforcing hardware control flow*

Over the years, a variety of mechanisms are provided in the literature to protect against buffer overflow but this attack still remains among the top vulnerabilities. In this section we provide a mechanism to prevent stack overflow attack

TABLE I: Buffer Overflow Exploits

Return Address Modification	Function Pointer Modification	Data Pointer Modification

`int func(char* user, int len) {`
`  char buff[100];`
`  memcpy(buff, user, len); //Vulnerability`
`}`
	`int func(char* user, int len) {`
	`  void (*fptr)(char *);`
	`  char buff[100];`
	`  fptr = &foo; //Addr. of desired function`
	`  memcpy(buff, user, len); //Vulnerability`
	`  fptr(user);`
	`}`
		`int func(char* user, int len) {`
		`  int *ptr;`
		`  int newdata = 0xaaaa;`
		`  char buff[16];`
		`  int olddata = 0xffff;`
		`  ptr = &olddata;`
		`  memcpy(buff, user, len); //Vulnerability`
		`  *ptr = newdata;`
		`}`

and format string attack by modifying the hardware of the processor. Our methodology involves implementation of a hardware stack which stores the function return address for each of the functions. In this scenario, even if the return address in the memory gets modified by the buffer overflow, hardware stack ensures that correct control flow of the program is maintained.

As an example, let us consider assembly code snippet of `stack.c` [5] in Table III:

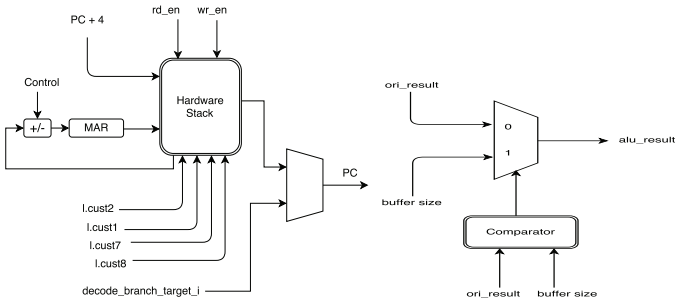
The code snippet shows the function prologue (when function arguments and return addresses are pushed on to the stack) and function epilogue (when return address is read off the stack).

In the OpenRISC architecture [4], register r_9, one of the 32 general purpose registers, acts as the link address which is used to calculate the return address of the function after execution completes. The register r_1 is the stack pointer which holds the limit of current stack frame. The register r_2 is the frame pointer which stores the address of the previous stack frame, (Table II [3] shows a function runtime stack).

Whenever a function is called, in OpenRISC architecture, it invokes either `l.jal` or `l.jalr` whereas `l.jr` is invoked to return to the called function. In this scenario, if buffer overflow occurs and the content of the r_9 is modified, then the code returns to a function with the modified return address. Now if the address provided by a malicious user causes buffer overflow to modify r_9 then the control flow gets transferred to the malicious code.

The operation of hardware stack is simple. Whenever it encounters a `l.jal` or `l.jalr` instruction, it pushes the next program counter value to the stack and alternatively if it encounters `l.jr` instruction with register r_9 as parameter, it pops its top value and passes that as the return address. Hence, even though the content of r_9 gets modified due to buffer overflow, the hardware stack remains immune against the attack and keeps on maintaining the correct control flow of the program. We enable the hardware stack by adding custom instruction `l.cust7`, which when enabled, the return address of the functions are read from the hardware stack.`l.cust8` freezes the Hardware Stack and `l.cust1` unfreezes it. Also we introduce another custom instruction `l.cust2` which disables the hardware stack.

There is an internal counter to keep track of the number of return addresses stored and also to update the message address register (MAR). At the software level, whenever a C code calls a main function the `l.cust7` instruction is followed by it. Before the exit of the main function the `l.cust2` instruction is executed. `l.cust8` is executed before the first `printf` statement and `l.cust1` before the last `printf` statement.



(a) Hardware stack (b) Secure `memcpy()`

Fig. 2: Hardware Modification

TABLE II: Memory Allocation inside OpenRISC

Position	Contents	Frame
FP+4N ... FP+0	Parameter N ... First stack parameter	Previous
FP-4	Return address	Current
FP-8	Previous FP Value	Current
FP-12 ... SP+0	Function variables ... Subfunction call parameters	Current
SP-4 SP-128	For use by leaf functions w/o function prologue/epilogue parameters	Future
SP-132 SP-2536	For use by exception handlers	Future

This ensures that the function return addresses are stored in the implemented hardware stack (Fig. 2a) which is secure from the buffer overflow attack.

B. Defeating Memory Corruption: Secure `memcpy()`

In the previous section, we have introduced the notion of hardware stack which prevents the return address modification. However, this approach does not prevent memory corrupting exploits. Hence, to protect against such vulnerabilities we introduced hardware enforced secure `memcpy()`. This protection prevents buffer overflow by hardware induced bound check and prevents any memory corruption due to buffer overflow. We will first present few observations on which we build our mitigation technique.

We have enlisted a part of the assembly code of `priv.c` [5] in the Table III. The `vuln` function starts with frame allocation and ends by jumping to its return address as mentioned in the previous section. In Table III we have included a part of the assembly which starts after the frame allocation till the `memcpy()` function is called. In line number 1 and 2 we store the input function parameters of the `vuln` function (stored in r_3 and r_4) to the local memory. It must be noted that during frame allocation $r_2 - 4$ and $r_2 - 8$ get reserved for previous frame pointer and return address of the `vuln` function. Hence, if any new variables are declared they will be stored after $r_2 - 8$ (r_2 is the current frame pointer). Further, new variables can be allocated either from top (from $r_2 - 12$) or from below ($r_2 - (frame\_size - 4)$). The code `priv.s` in Table III shows that the input function arguments are stored

TABLE III: Assembly code of priv.c and stack.c Exploits

priv.s	stack.s
vuln: l.sw -40(r2),r3 # SI store . . . l.lwz r4,-44(r2) # SI load l.addi r3,r2,-32 # addsi3 l.ori r5,r4,0 # move reg to reg l.lwz r4,-40(r2) # SI load l.jal memcpy # call_value_internal l.nop # nop delay slot	vuln: .LFB1: .cfi_startproc l.sw -8(r1),r2 # SI store . . . l.lwz r9,-4(r1) # SI load l.jr r9 # return_internal l.nop # nop delay slot .cfi_endproc

from below, whereas the new variables are stored from the top. This gives us a pretty straightforward approach to calculate the buffer size from the assembly itself. We keep a track of the location of the last variable declared. At this point, `memcpy()` function requires 3 arguments which are needed to be stored in registers r_3, r_4 and r_5. Among them r_3 stores the starting address of the buffer. By subtracting the address of the last variable from the starting address of the buffer we can get the buffer size. Now, all we need to do is to compare the computed buffer size with the parameter count and if the buffer size is less than the count we pass buffer size as an argument to `memcpy()` instead of `count`.

Considering `priv.s`, highlighted portion in the code snippet indicates the argument passed to `memcpy()` function. The first instruction ($l.addi\ \ r3, r2, -32$) transfers the starting address of the buffer ($r_2 - 32$) to r_3. The address of the latest variable in this case is $r_2 - 16$. Subtracting this two will give us buffer size which in this case is 16. The next instruction $l.ori$ transfers the function argument count to r_5 which denotes the number of memory locations to be updated by `memcpy()`. Now, we will check whether the instruction $l.ori\ \ r5, r4, 0$ returns the count value greater than the buffer size or not.

To achieve this, we introduce a set of instructions which can prevent and detect buffer overflow attacks. The operations performed by this instruction are listed below:

1) **l.cust3** This instruction will be inserted by the compiler just before `memcpy()` function is declared. It sets a specific flag inside the processor and observes the occurrence of `l.addi` and `l.ori` which are required for computation of buffer size. If the buffer size is less than the size of the argument count, a `smash_detect` flag is set and the value of the count argument is updated with the buffer size, ensuring both detection and prevention of buffer overflow.
2) **l.cust4** This instruction resets the `smash_detect` flag.
3) **l.cust5** This instruction induces a lock on latest variable address location to preserve it from intermediate function calls. This can be alternatively achieved by maintaining a hardware stack for latest variable locations for each function call.
4) **l.cust6** This instruction removes the lock.

An example of secure `memcpy()` function is given next:

Instruction `l.cust5` locks the latest variable address so that it does not get modified by the function call to printf. As we have mentioned this can be alternatively achieved by maintaining a hardware stack for this. `l.cust6` removes this lock and `l.cust3` ensures that `memcpy()` does not work for the buffer. This must be noted that our mitigation technique is essentially based on hardware modifications as we have developed and integrated new instructions for protection of memory corruptions and control flow violations. The placement of this instructions in the code can be handled by the compiler itself. We can create a new flag in the compiler which when enabled will force the compiler to implement the proposed instructions. Hence in this section we have shown how a hardware modification can protect against software based vulnerabilities without requiring any extra coding effort from the users.

IV. Result and Performance

We have prevented all the previously mentioned exploits by our modifications. In all these cases, the buffer overflow by the `memcpy()` function and return address modification by format string vulnerability have been prevented. Additionally, the `smash_detect` flag is observed though a `led`. The modified hardware takes around 15339 logic elements whereas the original hardware of *OpenRISC* takes around 11750.

V. Conclusion

In this work we have developed hardware based security policies to prevent buffer overflow attacks. Our proposal involves incorporation of new customary instruction in the OpenRISC architecture which when invoked, the compiler can ensure secure memory management. Essentially, we have protected `memcpy()` function which is widely used to trigger buffer overflows. Additionally, we have introduced a notion of hardware stack to protect against the return address modification. We have successfully prevented all the given exploits on Linux platform using secure `memcpy()` and hardware stack.

References

[1] Buffer overflow. https://en.wikipedia.org/wiki/Buffer_overflow.

[2] mor1k documentation. https://github.com/openrisc/mor1kx/blob/master/doc/mor1kx.asciidoc.

[3] Or1200 ip core specification. http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2For1200%2Fdoc%2Fopenrisc1200_spec.pdf&rev=645.

[4] Openrisc 1000 architecture manual. http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.1-rev0.pdf, 2014.

[5] Sample exploits for openrisc linux. https://github.com/nekt/csaw_esc_2016/tree/master/tools/exploits, 2016.

[6] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 189–200, 2012.

[7] Aleph One. Smashing the stack for fun and profit. http://phrack.org/issues/49/14.html, 1996.

[8] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Computers*, 55(10):1271–1285, 2006.

[9] Jonathan D. Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[10] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 1–17, 2016.

[11] Scut / Team Teso. Exploiting format string vulnerabilities. https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf, 2001.