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

alam.manaar@gmail.com, vidya.mazhur@gmail.com, sarani.bhattacharya@cse.iitkgp.ernet.in, deb.basu.roy@cse.iitkgp.ernet.in, rschakraborty@cse.iitkgp.ernet.in, debdeep@cse.iitkgp.ernet.in

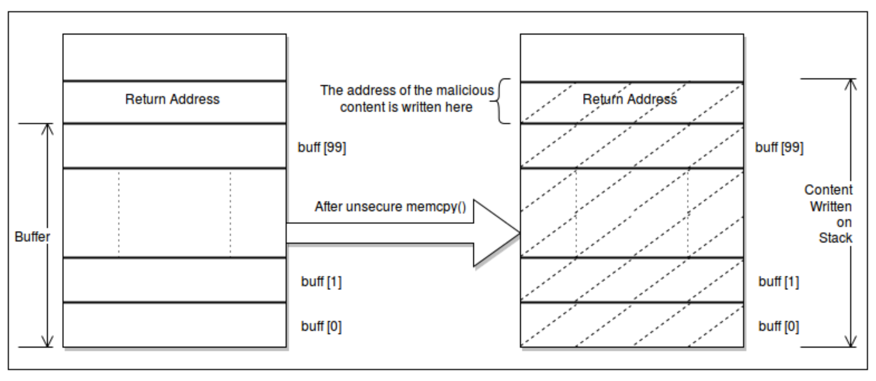
Secured Embedded Architecture Lab, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India

Introduction

- **Security threats to embedded systems**
  - Hardware and Software vulnerabilities
  - Performance-efficient languages such as C and C++ widely used for embedded applications
  - Vulnerable to memory corruption due to lack of secure management
- **Buffer Overflow: Trigger malicious code execution by overwriting correct memory content**
  - Software level countermeasures may be easily bypassed
  - Need hardware level countermeasures, e.g. hardware-based protection of the function return address
  - target platform for existing architectures different from the OpenRISC ISA processor

Attacking Control Flow

Return Address Modification




```
int func(char* user, int len) {
  char buff[100];
  memcpy(buff, user, len); //Vulnerability
}
```

Format String Vulnerability

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

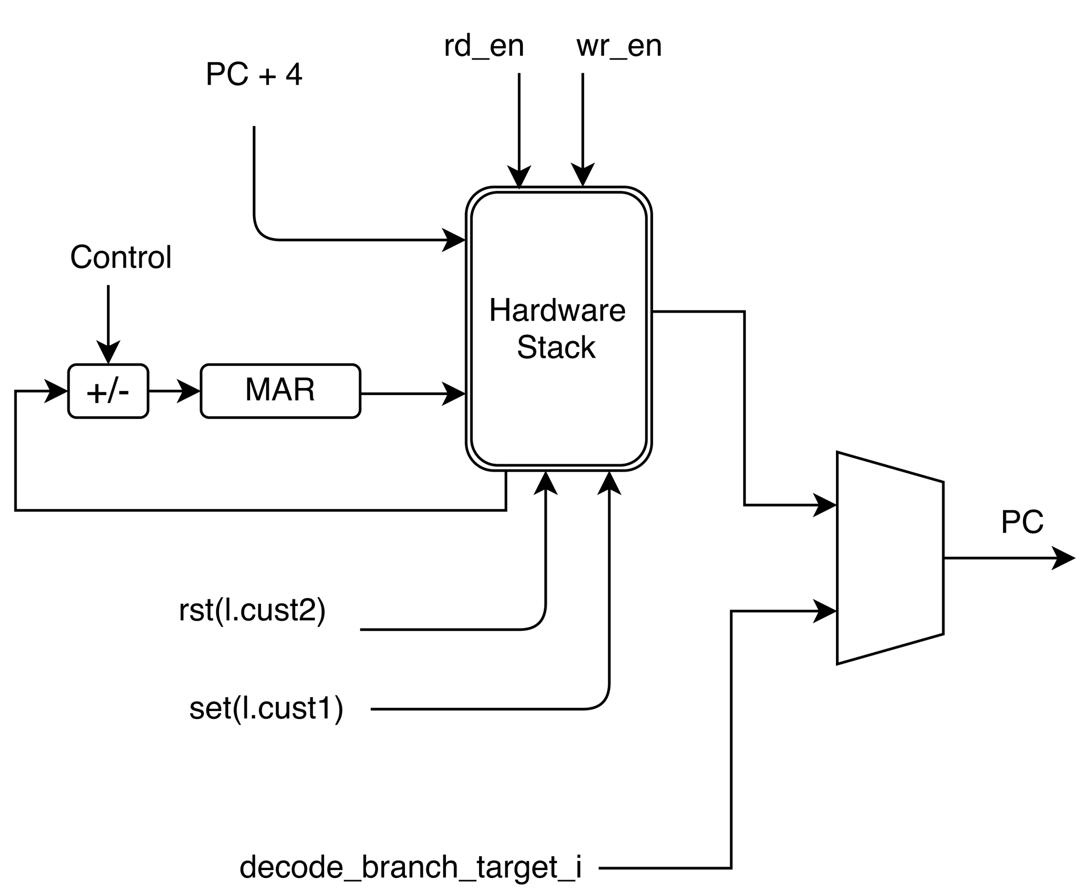
```
int func(char* user) {
  printf(user);  //Vulnerability
}
```

Example: Assembly Code for Stack.c

```
vuln:
.LFB1:
  .cfi_startproc
              .
              .
              .
  l.ori r1,r2,0 # deallocate frame
  l.lwz  r2,-8(r1) # SI load
  l.lwz  r9,-4(r1) # SI load
  l.jr   r9 # return_internal
  l.nop    # nop delay slot
  .cfi_endproc
```

- **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**

Our Hardware Stack



Our Objective

Hardware-Based Mitigation of Memory Corruption and Ensuring Control Flow Integrity for the OpenRISC ISA Processor

Our Contributions

- **Prevention of all forms of memory corruption and buffer overflow attacks on OpenRISC architecture**
- **Combination of compiler and hardware modification**
- **Introduction of new instructions via hardware modification for compiler to detect and prevent memory corruption via buffer overflow**

Protect Control Flow

- **Implementation of a hardware stack which stores the function return address for each of the function**
- Prevention using hardware stack:
  - Whenever it encounters a `l.jal` or `l.jalr` instruction, it pushes the next program counter value to the stack
  - 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
  - Custom instruction **`l.cust1`**, when enabled, ensures that the return address of the functions are read from the hardware stack.
  - Custom instruction **`l.cust2`** disables the hardware stack.

Prevent Mem. Corruption

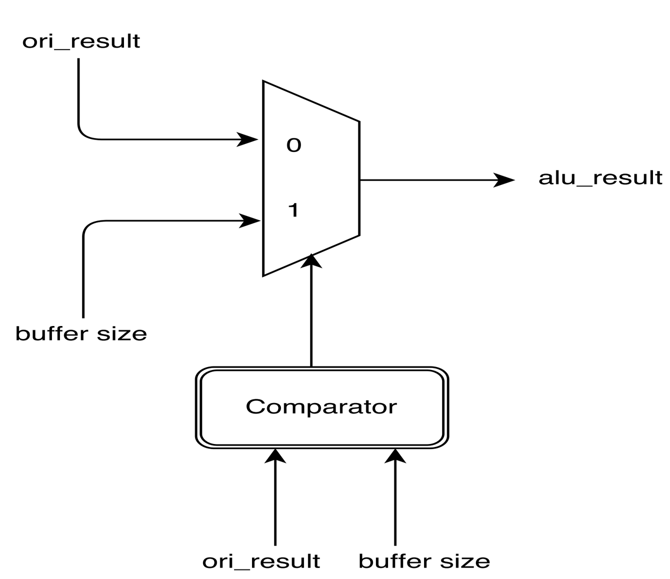
- **We introduced hardware enforced secure `memcpy`**
- This protection prevents buffer overflow by hardware induced bound check and prevents any memory corruption due to buffer overflow.

Example: Assembly Code for Priv.c

```
vuln:
l.sw -40(r2),r3 # SI store
  .
  .
l.sw -36(r2),r3 # SI store
  .
  .
  .
l.nop # nop delay slot
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
```

- 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 new 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.

Secure `memcpy`

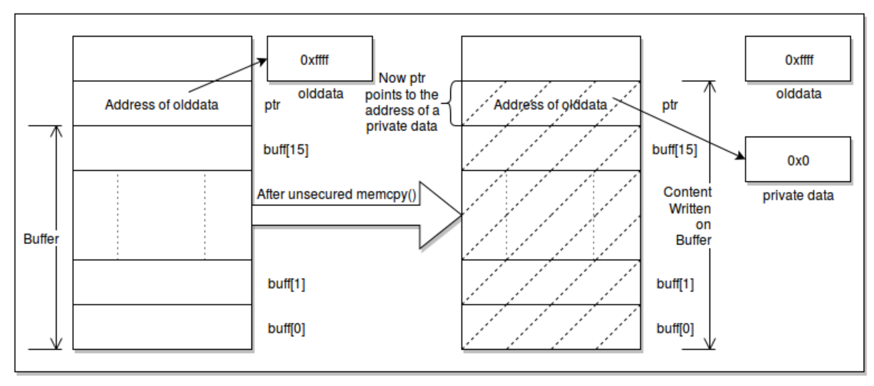


Overview

- The root cause of buffer overflow threat:
  - **`memcpy` does not impose any bound-checking during memory update**
- Our countermeasure approach:
  - **Introduction of new instructions that keep track of the buffer size**
    - Ensures number of memory locations upgraded by `memcpy` is less than or equal to buffer size
  - **Storing the return addresses in the hardware stack**
    - Prevention of return address modification
- Our results:
  - **Prevented `stack.c`, `ptr.c` and `priv.c` on Linux platform using our new instruction**
  - **Protection against return address modification by `format.c` using stack-based tracking of return addresses**

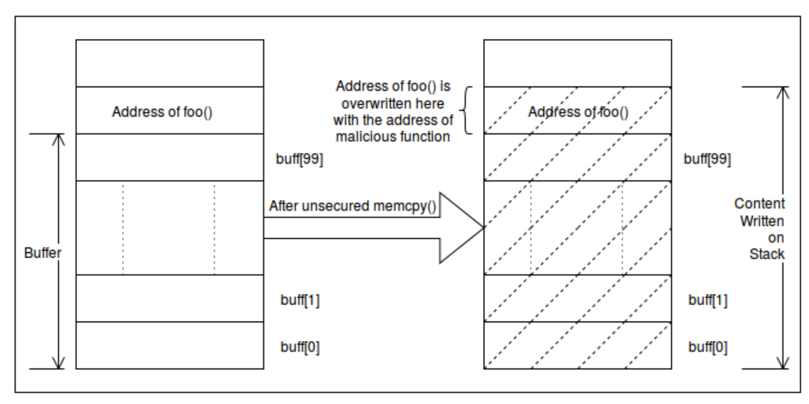
Memory Corruption

Data Pointer Modification




```
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;
}
```

Function Pointer Modification




```
int func(char* user, int len) {
  void (*fptr)(char *);
  char buff[100];
  fptr = &foo; //Address of intended function
  memcpy(buff, user, len); //Vulnerability
  fptr(user);
}
```

New Instructions

- **l.cust3** This instruction will be inserted by the compiler just before `memcpy` function is declared in C code to protect buffer overflow. This instruction 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 argument count a `smash_detect` flag is set and the value of the count argument is updated with the buffer size. Thus this instruction ensures both detection and prevention of buffer overflow.
- **l.cust4** This instruction resets the `smash_detect` flag.
- **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.
- **l.cust6** This instruction removes the aforementioned lock.