Customized Instructions for Protection Against Memory Integrity Attacks

Debapriya Basu Roy, Manaar Alam, Sarani Bhattacharya, Vidya Govindan, Francesco Regazzoni, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay

***Abstract*—Customized instructions have typically been used for enhancing the performance of embedded systems. However, the use of finding dedicated instructions for security has been rather limited. On the contrary, modern processors are crippled by the threats of memory integrity attacks, which typically target the control flow of a program and are mitigated at the software level. In this letter, we analyze the memory exploitation codes being developed as a part of the Cyber Security Awareness Week-2016 competition, which are based on unsecured `memcpy` and return address modification by buffer overflow on OpenRISC and RISC-V architectures, and implement protections at the hardware level. We added eight new instructions to handle the four exploits by designing dedicated hardware stack and a module for checking against buffer overflow. We have also performed a validation on RISC-V platform and introduced two new custom instructions to ensure security from unbounded `memcpy`. The proposed countermeasures and the new instructions are validated on field programmable gate array platform.**

***Index Terms*—Hardware stack, memory corruption, openRISC, RISC-V, Secure `memcpy`.**

I. Introduction

EMBEDDED electronic control units are an integral part of several critical infrastructures. They are often based on lightweight processors which run applications developed using high-level languages, like C and C++, which do not have explicit integrity checks for memory operations. The accompanying compilers are also not adequately smart to detect these issues, which can lead to severe attacks. Buffer overflow is still one of the major concerns in developing secured applications, and are based on the absence of proper memory boundary checks during function calls, functions returns, variable access, etc. Software level countermeasures though claim to prevent memory corruptions; they can itself be bypassed by advance malwares. Vis-à-vis, protections through hardware are comparatively more secure as they enforce checks at the physical level.

Manuscript received March 15, 2018; accepted April 10, 2018. Date of publication April 18, 2018; date of current version September 7, 2018. This manuscript was recommended for publication by M. Maniatakos. *(Corresponding author: Debapriya Basu Roy.)*

D. Basu Roy, M. Alam, S. Bhattacharya, V. Govindan, R. S. Chakraborty, and D. Mukhopadhyay are with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur 721302, India (e-mail: dbroy24@gmail.com).

F. Regazzoni is with the ALaRI Institute, University of Lugano, 6900 Lugano, Switzerland.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2018.2828506

A. Related Work

A hardware-based Watchdog for protection against memory integration violation was proposed in [1] to develop a unique identifier at the hardware level for allotment of each memory location. When this memory location is reaccessed, it checks the validity of the generated identifier and detects if any abnormality exists. In [2], the idea of hardware stack was introduced to defend against the attacks which trigger malicious code execution by modifying the function return address. During return statement, if the popped value from the hardware stack does not match with the popped value from the program stack, the processor alerts the user. Further, in [3], hardware-assisted data-flow isolation mechanism was proposed to ensure protection against memory integrity attacks. In this case, an extra tag was attached to every memory unit to protect any malicious memory update. However, in both of these work, the implementation platform is not similar to our scenario and thus cannot be applied directly in our case.

The objective of this letter is to integrate specific customized instruction into the processor architecture to prevent memory integrity violation attacks. More specifically, we concentrated on two embedded processor architectures, OpenRISC and RISC-V. We introduced eight different instructions in the OpenRISC architecture to protect against vulnerabilities of return address modification and insecure execution of `memcpy` function. Additionally we have shown that though RISC-V architecture can prevent return address modification through tagged memory, it is still vulnerable against insecure execution of `memcpy`. Henceforth, we extended the RISC-V instruction set to introduce two new instructions which can prevent the aforementioned vulnerability. The overhead of the proposed countermeasures is found to be 30% and 17% for OpenRISC and RISC-V architecture, respectively. A preliminary version of this letter has been published in [4]. In this letter, the focus is more on the RISC-V architecture significantly extending the work presented in [4].

II. Buffer Overflow in OpenRISC

The theme of embedded security competition of the Cyber Security Awareness Week-2016 (CSAW-2016) event was buffer overflow attack. We were provided with four different C codes which triggered various malicious routines using buffer overflow [5]. Details of these exploitation codes and methodologies behind the attacks are discussed in details in [4]. These exploit are summarized in Table I.

To protect against these vulnerabilities, we need to prevent the modification of return address through hardware-enforced control flow and avoid the insecure execution of `memcpy` function. Next, we define the threat model in details.

TABLE I
DESCRIPTION OF EXPLOIT CODES

Exploits	Objective	Methodology
Stack.c	To modify the return address of function to execute a malicious function	Return address modification by `memcpy` function
format.c		Return address modification by unsanitized printf
ptr.c		Modifying function pointers by insecure `memcpy` function
priv.c	To modify the data pointers to change its value without any explicit access	Data pointer modification by insecure `memcpy` function

TABLE II
SUMMARY OF PROPOSED NEW INSTRUCTIONS

Return address modification prevention through hardware stack	l.cust7	Allows processor to read the return address from the hardware stack.
	l.cust8	Freezes the hardware stack, preventing any further push operation.
	l.cust1	Unfreezes the hardware stack.
	l.cust2	Disables the hardware stack.
Securing `memcpy()` through hardware enforced bound check	l.cust3	Enables the hardware module to compute the available memory blocks available for the destination array. Also sets a `smash_detect` flag if buffer overflow is detected.
	l.cust4	Disables `smash_detect` flag.
	l.cust5	Locks the latest variable address location to prevent any update due to intermediate function calls.
	l.cust6	Removes the aforementioned locking mechanism.

A. Threat Model

In this letter, we propose a generic architectural solution against buffer overflow without requiring any intervention by operating system or compiler. To do so, we propose custom instructions which check for unbounded memcpy instructions and automatically react to this situation preventing the malicious code from mounting a successful attack. Executions of these instructions are done inside a critical section, where these instructions are allowed to be enabled or disabled either in a pair or they are denied the permission to execute. For example, in our proposed hardware stack solution, enabling and disabling of the hardware stack are initiated by two customized instructions. However, disabling of hardware stack can be done only from the function in which enable instruction of hardware stack is executed. These prevents usage of these instruction inside a malicious function like `memcpy`.

B. Hardware Integration

This section discusses the integration of our customized instructions in embedded processor architecture. Exploits which corrupts memory can be prevented by introducing hardware-enforced bound check which will secure functions like `memcpy`. On the other hand, exploits, which corrupt control flow of a program, can be prevented by keeping a copy of the function return address in a hardware stack which a user process cannot access. To integrate these countermeasures, we have added eight new instructions to the OpenRISC architecture. Table II summarizes these eight instructions. Next, we present a brief description of their operational principles.

1) Implementing Hardware Enforced Control Flow: To ensure a hardware-enforced control flow, we have implemented a hardware stack which, when enabled, keeps a copy of the function return address for each function call. The hardware stack can be controlled by four instructions. The `l.cust7` instruction enables the hardware stack and forces the processor to read the return address from the hardware stack rather than the return address register r_9. Hence, in this scenario, modifying the content of r_9 is of no use as the hardware stack does not get edited by an eventual stack overflow and thus keeps track of the correct control flow of the program. The architectural diagram of hardware stack is shown in Fig. 1(a).

TABLE III
EFFECT OF THE COUNTERMEASURE ON CSAW EXPLOITS

	stack.c	format.c	ptr.c	priv.c
Hardware Stack	✓	✓	✓	×
Secure memcpy	✓	×	✓	✓

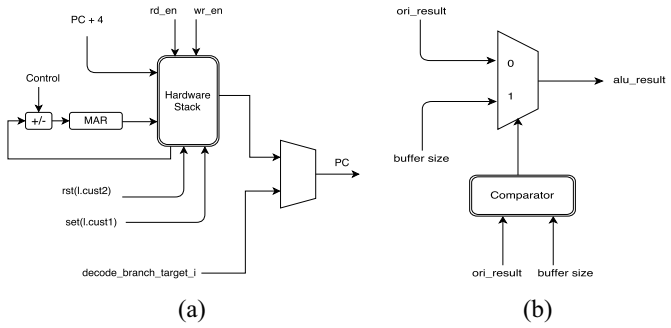

Fig. 1. Hardware modification. (a) Hardware stack. (b) Secure `memcpy()`.

2) Securing Memory Access (Secure `memcpy`): To prevent attacks based on insecure `memcpy`, we propose hardware-enforced secure `memcpy` [Fig. 1(b)]. Our secure `memcpy` implement hardware-enforced bound check to protect against buffer overflow. Our strategy is to store the location of the last variable declared in a specific register. The `memcpy` function requires the content r_3, r_4, and r_5 registers, as these three registers store the function's argument values. Register r_3 stores the starting address of the buffer. Hence, we can easily compute the buffer size by subtracting the address stored in the register r_3 from the address of the last variable. If the buffer size does not match with the parameter `count` of the `memcpy` function, we use the available buffer size as an argument to `memcpy` function rather than `count`. The effect of these countermeasures on the exploit codes are summarized in Table III, while a complete description is reported in [4].

A user can use these instructions to activate the countermeasure to prevent buffer overflow. In our threat model, we assume that the supplied libraries to the user may have some insecure features which can be exploited by a malicious adversary. The proposed new instructions can be placed at appropriate position before the execution of the supplied library function to ensure the integrity of the control flow of the program. An example of such is shown in the following code snippet.

```
asm volatile("l.cust5");
printf("vuln() has received %d bytes\n", count);
asm volatile("l.cust6");
asm volatile("l.cust3");
memcpy(buff, s, count);
```

C. Result and Performance

We have integrated the proposed countermeasures and new instructions in the OpenRISC processor and have implemented it on DE0-NANO board. We have successfully detected and prevented the buffer overflow attack for each of the given exploits of CSAW competition [4]. The modified processor architecture of OpenRISC occupies 15339 logic elements, whereas the logic element requirement of the original OpenRISC architecture is 11750. The critical path of the

```
void foo (char *bar){
  int My_Int = 10;char c[28];
  printf("My Integer value before memcpy= %x\n", My_Int);
  memcpy(c, bar, strlen(bar)); // no bounds checking...
  printf("My Integer value after memcpy= %x\n", My_Int);}
int main (int argc, char **argv){
  char bar[32]= "my string is too long!!!\x10\x10\xc0\x42\x10\x10\xc0\x42";
  foo(bar);return 0;}
```

Code 1. Buffer overflow in RISC-V.

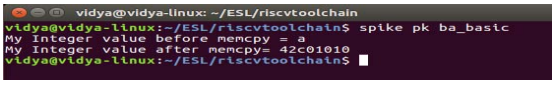

Fig. 2. Illustration of vulnerability due to insecure `memcpy`.

OpenRISC processor does not change after the integration of the proposed countermeasures. Additionally, as we have shown in [4], we need to execute four extra instructions to ensure secure `memcpy` or to use the proposed hardware stack. Thus, to prevent the malicious code execution, the overhead is the execution time of this four extra instruction.

III. Validation on RISC-V

We extended the evaluation of the instructions for protection in RISC-V. Our RISC-V platform is based on Pulpino architecture [6] which is utilized in the Shakti processor [7]. This processor core implements the base RISC-V instruction set with some additional instructions for efficient implementation of post-increment, multiplication, and accumulation.

RISC-V architecture is an upgrade of OpenRISC instruction set. Henceforth in RISC-V, designers have already integrated countermeasures to prevent malicious execution through buffer overflow attacks. The exploits provided in the CSAW competition either try to modify the return address of the function or try to take advantage of the insecure `memcpy` function. However, RISC-V architecture prevents any malicious modification of return address through the `tagged memory` countermeasure [8]. When tagged memory is used, the architecture maintains a `tag` bit which can only be set when the return address is modified through the function calls. Any other modification (due to buffer overflow or due to a bug) of memory locations which stores the value of return address will reset the `tag` bit. During the return call, the architecture will interrupt the program execution if the `tag` bit is not set, thus preventing any malicious code execution.

RISC-V architecture though provides protection against malicious modification of return address value, it is still vulnerable to attacks which exploit insecure `memcpy` functions. We illustrate this vulnerability with a simple code snippet 1.

This example shows the threat of insecure execution of `memcpy` function. The objective of this code is to modify the value of the variable `My_Int` without even accessing it. This is achieved by inducing buffer overflow through `memcpy` function which does not offer any bound check. As it is shown in Fig. 2, that objective is achieved.

This threat can be mitigated by using a secured `memcpy` similar to the one we proposed for the OpenRISC architecture. However, due to the difference between the ISA of OpenRISC and RISC-V, the exact implementation of the secure `memcpy` will be different. We will explain the details of the implementation starting from the assembly realization of the attack routine presented in Code 1.

The assembly routine starts with the initialization of the stack pointer. Then the value of the return address register

```
foo:
  addi sp,sp,-64 ; sd ra,56(sp) ; sd s0,48(sp) ;
   addi s0,sp,64 ; sd a0,-56(s0) ;
   li  a5,10 ;
   sw  a5,-20(s0) ;
   lw  a5,-20(s0) ; mv a1,a5 ;
   lui a5,%hi(.LC0) ; addi a0,a5,%lo(.LC0) ; call printf ;
   ld  a0,-56(s0) ; call strlen ; mv a4,a0 ;
   addi a5,s0,-48 ;
   mv  a2,a4 ; ld a1,-56(s0) ; mv a0,a5 ; call memcpy
```

Code 2. Assembly code of Code 1.

TABLE IV
Memory Allocation Inside RISC-V

Position	Content			
$sp+60$	Return address $(ra[hi])$			
$sp+56$	Return address $(ra[low])$			
$sp+52$	Frame Pointer $(s0[hi])$			
$sp+48$	Frame Pointer $(s0[low])$			
$sp+44$	$a_5 = 10$			
$sp+40$	c[27]	c[26]	c[25]	c[24]
$sp+36$	c[23]	c[22]	c[21]	c[20]
⋮	⋮			
$sp+20$	c[7]	c[6]	c[5]	c[4]
$sp+16$	c[3]	c[2]	c[1]	c[0]

(*ra*) and frame pointer register (*s*0) are stored at the appropriate location and the frame pointer register is updated with value $sp+64$. Consequently, memory is allocated for variable `My_Int` at $sp+44$. Finally, function calls corresponding to `printf, strlen`, and `memcpy` are performed. In our analysis, we ignore the function call of `printf` and `strlen` and concentrate on the execution of `memcpy` function. RISC-V architecture provides 6 registers ($a0-a5$) which are used to pass the arguments of the functions. `memcpy` function requires three arguments which are stored in registers $a0$, $a1$, and $a2$. The value stored in $a0$ and $a1$ indicates the starting address of destination (`c`) and source (`bar`) arrays, respectively, whereas $a2$ indicates the number of characters to be copied into the destination array from source array. As visible in Code 2, the starting address of the character array `c` is computed by instruction `addi a5,s0,-48` This means that the starting address of the character array `c` is $s0-48 = sp+64-48 = sp+16$, whereas the variable `My_Int` is stored in location $sp+44$. It must be noted that in RISC-V architecture the address values are 64 bits. The integer size 32 bit and a character size is 8 bit. Additionally, RISC-V supports byte level addressing. Using all these information, we were able to construct the memory allocation table as shown in Table IV. From Table IV, we see that when we try to copy array `bar` into array `c`, we have an overflow, since the size of `bar` is larger than size of `c`. This overflow eventually modifies the value of the `My_Int` variable. To protect from this unbounded memory copy, we enforce a hardware induced bound check of the source and destination arrays. We devise a method to count the number of memory blocks allocated for destination arrays. In Fig. 3, we show the architectural block diagram of the proposed countermeasure.

RISC-V natively supports the addition of new instructions. We integrated our instructions using this existing support. To prevent the buffer overflow vulnerability we have introduced two new customized instructions in the RISC-V ISA. The first instruction (`lr.cust1`) will set the `activate` flag (Fig. 3) which in turn will set the `cust_inst_en` signal. This is a control signal which makes the other modules of the countermeasure active. Now, we first calculate the available memory blocks for the destination array. This can be computed by observing the starting address of the destination array and the address of the last variable declared. The address of the last

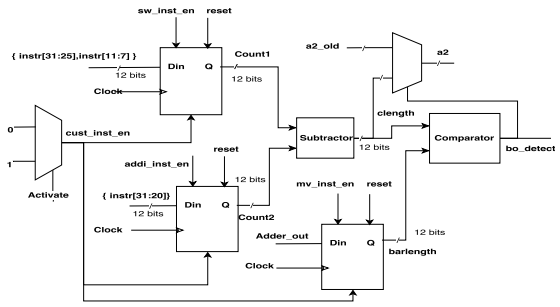

Fig. 3. Execution of the proposed countermeasure.

```
void foo (char *bar){
  asm volatile( "fence");asm __volatile__ (".word 0x02C5856B\n");
  int My_Int = 10;char c[28];
  printf("My Integer value before memcpy= %x\n", My_Int);
  memcpy(c, bar, strlen(bar)); // no bounds checking...
  asm volatile( "fence"); asm __volatile__ (".word 0x02C5852B\n");
  printf("My Integer value after memcpy= %x\n", My_Int);}
```

Code 3. Protected C code in RISC-V.

variable declared can be found by observing the instruction `sw a5,−20(s0)`, whereas starting address of the destination array can be found by observing `addi a5,s0,-48`. We calculate the available memory blocks of destination array by subtracting the immediate offset value of the aforementioned instructions. Once the value of available memory blocks is calculated, we compare it with the third argument of the `memcpy` function, which indicates the number of character elements to be copied. The value of this argument get stored in the *a*2 register through `mv a2, a4` instruction. It must be noted that there is no explicit `move` instruction in RISC-V ISA. Rather, `mv,a2,a4` gets implemented by executing `addi a2,a4,0`. If the value of the third argument is larger than the available memory blocks of the destination array, we set the `bo_detect` flag indicating the occurrence of buffer overflow. We change the value of the *a*2 register with the size of the destination array if `bo_detect` flag is set.

Code 3 shows the protected C code which incorporates the countermeasure discussed above. We have added two custom instructions `cust1` and `cust2` with opcodes `0x02C5856B` and `0x02C5852B` using the assembly extension in the C code. The fence instructions are added before the assembly extensions to guarantee ordering between memory operations. The signal `activate` goes high whenever the `cust1` instruction is encountered and it is pulled down to low when the `cust2` instruction is encountered.

A. Overhead and Result

We implemented RISC-V architecture on Kintex-7 device available in Sakura-X board using Xilinx's Vivado 2016.1 version. To integrate the proposed countermeasures, we modified the `decode` section of RISC-V architecture. The corresponding overhead only for the `decode` module is shown in Table V. Table VI gives the resource utilization for the entire RISC-V architecture which shows an increase in look up table, look up table based RAM, flip-flops, block RAM, BUFG, and input-output pins resources by 4.66, 56.4, 14, 0.3, 0.5, and 53.3%, respectively, in the countermeasure design. Overall the overhead on the entire RISC-V architecture is approximately 16%. Critical path of RISC-V processor does not change after the integration of the countermeasure and we need execution of two extra instruction to prevent the execution of insecure `memcpy` as shown in Code 3.

TABLE V
RESOURCE UTILIZATION WITH AND WITHOUT COUNTERMEASURE FOR DECODE LOGIC OF RISC-V ARCHITECTURE

Resource element	Without countermeasure	With countermeasure	Overhead (%)
LUT	937	987	5
LUTRAM	308	308	0
FF	9	62	85
IO	970	982	1.2
BUFG	1	1	0

TABLE VI
RESOURCE UTILIZATION WITH AND WITHOUT COUNTERMEASURE FOR COMPLETE RISC-V ARCHITECTURE

Resource element	Without countermeasure	With countermeasure	Overhead (%)
LUT	52556	55125	4.66
LUTRAM	308	708	56.4
FF	21477	24981	14
BRAM	193.5	194	0.3
DSP	24	24	0
IO	7	15	53.3
BUFG	2	4	0.5

IV. CONCLUSION

In this letter, we focused on hardware-enforced security strategies to ensure detection and prevention of buffer overflow attacks. We have introduced eight new customized instructions in the OpenRISC instruction architectures. These instructions, when applied, can prevent any violation of memory integrity and program control flow by protecting insecure memory access and return address modification. The user can instantiate these instructions without requiring any compiler or linux kernel modification. We have prevented all the exploits, provided in CSAW competition, successfully on the given Linux platform. We have also shown that the RISC-V architecture is vulnerable to buffer overflow attack as it does not enforce any bound check during the execution of the insecure `memcpy` function. The proposed countermeasure, which ensure secure execution of `memcpy` function, can also be applied on the RISC-V architecture with little modification. The overhead of the proposed countermeasures on the OpenRISC and RISC-V architectures has been provided and, considering the colossal threat of buffer overflow, the overhead is not significant.

REFERENCES

[1] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. ISCA*, Portland, OR, USA, 2012, pp. 189–200.

[2] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A hardware solution to prevent security attacks on the function return address," *IEEE Trans. Comput.*, vol. 55, no. 10, pp. 1271–1285, Oct. 2006.

[3] C. Song *et al.*, "HDFI: Hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, May 2016, pp. 1–17.

[4] M. Alam *et al.*, "SmashClean: A hardware level mitigation to stack smashing attacks in OpenRISC," in *Proc. MEMOCODE*, 2016, pp. 1–4.

[5] *Sample Exploits for OpenRISC Linux*, Accessed: Aug. 3, 2018. [Online]. Available: https://github.com/nekt/csaw_esc_2016/tree/master/tools/exploits

[6] A. Traber *et al.*, "PULPino: A small single-core RISC-V SoC," in *Proc. RISCV Workshop*, 2016. [Online]. Available: http://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf

[7] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T: A RISC-V processor with light weight security extensions," in *Proc. ACM HASP*, Toronto, ON, Canada, 2017, pp. 1–8.

[8] P. Jantscher, M. Werner, and S. Mangard. *Counteracting Code-Reuse Attacks Using Tagged Memory on a RISC-V Architecture*. Accessed: Aug. 3, 2018. [Online]. Available: https://diglib.tugraz.at/download.php?id=5891c8d7bc3b4&location=browse