http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 13 (7): 1351-1355, 2014
ISSN 1812-5638 / DOI: 10.3923/itj.2014.1351.1355
© 2014 Asian Network for Scientific Information

A Technique Against Buffer Overflow Attacks for Embedded Systems via., Hardware/Software

[1,2]Wei Guoheng and [2]Li Zheng
[1]School of Computer, Wuhan University, Hubei, China
[2]Department of Information Security, Naval University of Engineering, Hubei, China

Abstract: Buffer overflow attacks cause serious security problems in embedded systems. The program presents an integrated software hardware approach to protect against buffer overflow attacks. This technique does not change the processor core, but instead adds a FPGA module that sits between cache and memory and that is able to defend return addresses from buffer overflow attacks. The solution exhibits neither the performance overhead of software solutions nor the CPU redesign costs of hardware solutions. The experimental results show that this strategy is effective.

Key words: Buffer overflows attack, computer architecture, embedded design

INTRODUCTION

Embedded Systems is the most popular areas nowadays and the momentum of rapid development has attracted the attention of the people from all walks. Embedded systems are application-centric, based on computer technology. The software and the hardware can be cut and a dedicated computer system is strict requirements for functionality, reliability, cost, size, power consumption (Zhang, 2008). Embedded system has the following characteristics, there are strict requirements of embedded real-time systems, there are very high power consumption and reliability requirements; there are some limits by resources and costs in the design process; there are some limits by the size and weight (Zhao *et al*., 2008).

With the widespread use of embedded systems, the security issues have become increasingly prominent. The currently embedded systems subject to the following aspects threat; first, a wide range of software attacks which depending on the attacker sufficient privileges in the execution environment, in order to be able to control and access to sensitive devices and data; the second is the widespread use of a data structure of predatory attacks, such as the buffer overflow attacks, third, someone will put implantation codes into the system, forth, it is called denial of service attacks (Frank and Bruno, 2008).

At present, the attacks on the buffer most affected. In some cases, the data exceeds the buffer storage area which is allocated, because the excessive return address points to the attacker's malicious code. When the function tries to return, they begin to perform code which hackers write in the wrong return address. If this hazard is reached, the intruder will gain the direct control of the system.

Many experts and scholars carry out a study on how to protect the embedded system to against buffer overflow attack. For example, Shao *et al*. (2006) proposed a combination of hardware and software technology, Wan *et al*. (2011) build a defense mechanism based on Fine-grained Instruction Flow Monitoring (FIFM). Du *et al*. (2006) proposed the establishment of a new embedded security architecture of network security devices and mechanisms for multi-level anti-buffer overflow attacks. Domestic longxin uses a special hardware mechanism to limit the value of the stack segment to solve this problem (Hu and Tang, 2003).

In this program, the means of defense, the use of software and hardware technology monitor and protect the role of the function's return address to prevent buffer overflow attacks. The experimental analysis are verified and confirmed its effectiveness which is based on the design discussed.

PROGRAM STRUCTURE

The program is a combination of hardware and software technology to monitor and protect the function's return address. Due to the introduction of the hardware module, the kernel of the processor does not need to be redesigned and it will not affect the performance of the software of the system. Even if the memory implanted attack code, as long as the return address is not tampered with, then the attack code will not trigger.

Corresponding Author: Wei Guoheng, School of Computer, Wuhan University, Hubei, China

At the hardware level, between the cache and main memory add hardware security module guard. The module not only can select a programmed logic device which can increase the efficiency of the CPU and can be configured but also can choose a simple connection between the CPU and the system bus gateway chip. The program selected FPGA as the guard module to increase the security module which processor architecture is shown in Fig. 1.

At the software level, the compiler inserts a new command before every function call and return instructions. The new directive is used to complete the operation about guard interface.

The program flow is as follows. The return address is first copied to the memory stack RA guard and the memory area can't be any direct calls for memory access. Guard stack of RA by the modified compiler is responsible for maintenance and management. New command stack RA state and the state of the program stack are synchronization and is independent of the location of the code and cache strategy. The return address will be stored in the stack RA in every function call; when the function returns, the guard will have buffer overflows checked to make sure that the correct return address has been returned by the processor. Therefore, this program don't care about the buffer overflow occurs or not. he guard maintained as long as the return address correctly and effectively, so the flow of execution of the program is not destroyed.

The program despite an increase in the guard module and expanded instruction set, but it don't bring too much overhead. First, the modification of the source code is changed at compile tool chain stage before the CALL and RETS instruction, so the amount of the operation of the entire program and did not add much. Followed by the guard module is located between the cache and main memory. It does not change the overall structure of the processor and does not affect access to the memory.

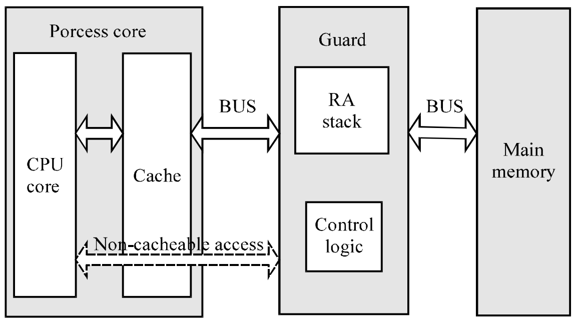

Fig. 1: Increased processor architecture of guard module

DESIGN DETAILS

The protection of the return address can be briefly summarized as two points: First, the return address is stored in the stack RA when function calls; second, stack RA return accurate address when the function returns.

Therefore, the key of the program is that the guard module must be able to distinguish every function call and return. The compiler adds a memory-mapped instruction "push_guard" before a CALL instruction by configuring. The function's return address is pushed into the guard module stack RA. Similarly, the Instruction "guard_attention" will be inserted into before each RETS instruction to inform the guard to verify protection of the buffer. The detailed algorithm of instruction “push_guard” and “guard_attention” is given in Fig. 2 and 3, respectively.

The address (RET_Addr) into the guard, “push_guard” will put the correct return upcoming RET_Addr to be pushed onto the stack RA. After that, guard detect stack RA boundary and store layer function's return address in the main memory. Finally pop_ret_addr load into the processor return address register. About the address pop_ret_addr discussion will be given in the following sections.

Guard_attention notify the guard module before the function returns and the instruction requires pointer register coordinate with the PC and FP. And ready for a branch instruction Branch_RET_Addr, the program will jump to the correct return address if the processor executes the instruction. The guard module contains the stack RA which store return address and control logic

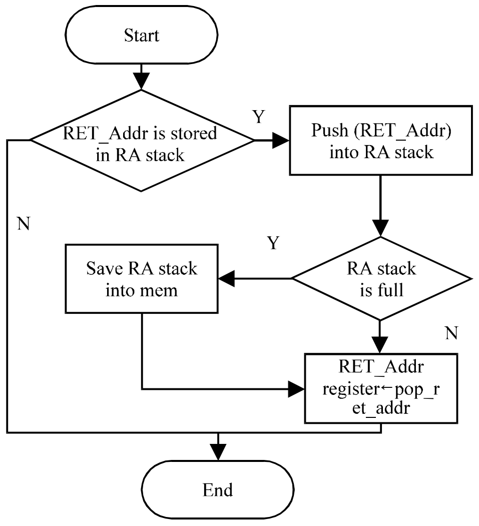
Start
RET_Addr is stored in RA stack
Y
Push (RET_Addr) into RA stack
N
Y
Save RA stack into mem
RA stack is full
N
RET_Addr register←pop_ret_addr
End

Fig. 2: Algorithm flow chart of push_guard

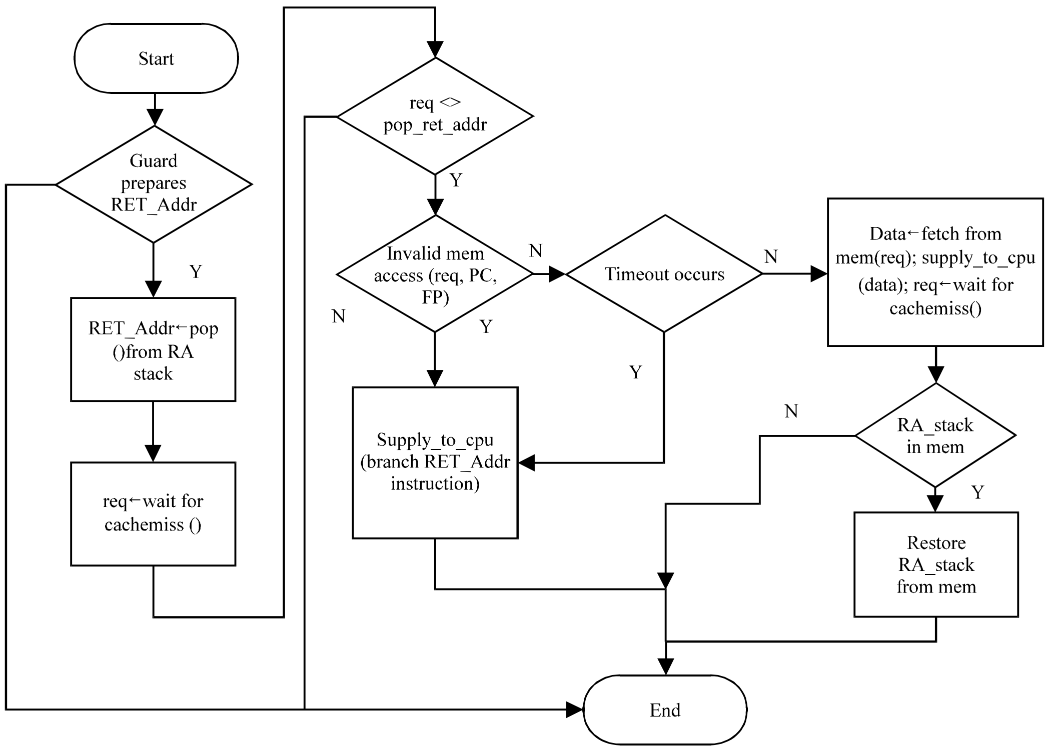
Start
Guard prepares RET_Addr
Y
RET_Addr←pop ()from RA stack
req←wait for cachemiss ()
req <> pop_ret_addr
Y
Invalid mem access (req, PC, FP)
N
Y
N
Timeout occurs
N
Y
Supply_to_cpu (branch RET_Addr instruction)
Data←fetch from mem(req); supply_to_cpu (data); req←wait for cachemiss()
RA_stack in mem
N
Y
Restore RA_stack from mem
End

Fig. 3: Algorithm flow chart of gurad_attention

unit which is used to verify the address of the instruction to determine whether the buffer attacked. When a cache miss occurs, the processor will be released after the guard of the memory instruction request. The new instructions are executed form of storage mapping. The address of the non-cacheable is pop_ret_addr. It is triggered guard work when the processor accesses the address. Follow the process operation to ensure the implementation of the protection program:

- Call push_guard instruction before the function happen By modifying and the real return address store in the guard module stack RA
- Select non-cacheable address (pop_ ret_addr) and writes it into the returns the address of the register. The address points to the original return address
- Call guard_attention notice to the guard ready before function returns
- Within the time threshold, CPU will distribute pop_ret_addr content request and it will some missing in the cache(If the valve is over time that will be thought under attack and branch command is executed)
- Guard analyzes the request, the branch RET_Addr instruction will be injected into the CPU if it determines that the attack occurred. Jump to RA maintains the correct address from the stack before the function returns

EXPERIMENTAL ANALYSIS

In order to verify the effectiveness of the program and test the system overhead of the time, Use simplescalar (Austin *et al*., 2002) simulation test kit on ARM development board do some test. Testing MiBench (Guthaus *et al*., 2001) contains six benchma-rks: bitcount---testing the bit processing capabilities of processor; crc-cyclic redun-dancy check; dijkstra---algorithm to find the shortest path between two nodes in figure; fft---transform algorithm for the frequency domain analysis of the digital signal processing; sha---A standard hash algorithm in the secure computing; stringsearch---string search algorithm. Every function call guard module will detect and prevent, therefore, the overall system performance overhead largely depends on the frequency of function calls in the program. The six benchmark program used in the test function call in each of 100 clock cycles occur the frequency as that bitc, crc, dijkstra, fft, sha and stringsearch is 0.7, 3.0, 0.12, 1, 0.12, 1, 0.6, 0.5.

The overhead P_C of one function call required is as follows:

$$P_C = T_G+T_S$$

Among them, T_G is on behalf of the guard access time and T_A is on behalf of the guard of access time to stack R_A. Specific time is determined by the CPU frequency and the guard module operating frequency and bus access speed. In general, a T_G need for a bus clock and a 5 guard clock; T_S generally consume a guard clock. A function of system overhead P_R returns as follows:

$$P_R = T_G+T_V+T_S$$

And the same time, T_S and T_G don't change and increase the verification of guard module for the return address T_V. T_V consumes a guard clock. Therefore, the implementation of the program increase the total overhead as follows:

$$P = I+C_m+(Nc\times Pc)+(N_R\times P_R)$$

where, I refer to the consumption of the increased instruction execution time. C_m refers to cache misses of the time. N_c means umber of function calls. N_R means the number of function returns.

The test system based on embedded platform, using the 32 bit microprocessor, the CPU clocked at 400 MHZ, equipped with FPGA (guard module) operating frequency of 200 MHz.The transmission frequency of the bus and storage memory access frequencies are 100 MHZ. Nearly guard access requires two CPU clock, a bus cycle cost 4 CPU clock. Simulation results are shown in Fig. 4 by simoutorder simulation of The SimpleScalar simulator. The average overhead of test time is 8%.But CRC is a special case. The function call frequently resulted in about 27% of the cost of the program and it does not have a greater impact on system performance.

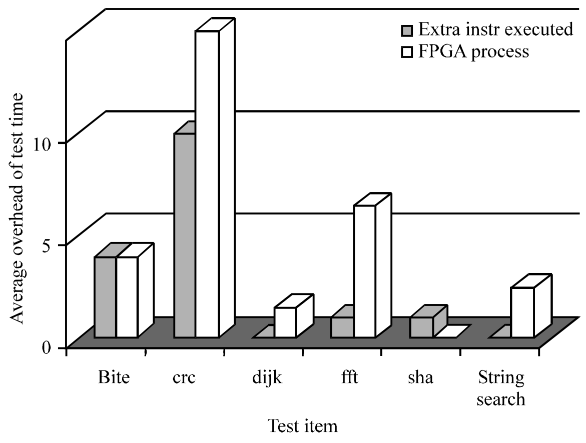

Fig. 4: Average overhead of test time

CONCLUSION

In this program, buffer overflow attacks are given and validated a combination of hardware and software protection scheme. The program has the following advantages:

- When hardware modules monitor the buffer attack protection, there is no need to modify the CPU or the ISA
- It does not take up too much CPU resources, because the return address verification protection occurs in the guard
- Modify OS Minimally (only call or ret)
- For embedded systems

The tests proved that the combination of hardware and software buffer-overflow exploit prevention program is an effective defense mechanism. However, this article only discussed in the stack smash attacks in the program. The next step will be for the rest of the buffer overflow attacks the optimization and upgrading of the program to truly improve computer systems, especially embedded systems against buffer overflow attack capability.

REFERENCES

Austin, T., E. Larson and D. Ernst, 2002. SimpleScalar: An infrastructure for computer system modeling. Computer, 35: 59-67.

Du, J., J. Jin and G. Li, 2006. Protection against buffer overflow embedded network devices Comput. Eng. Des., 27: 2918-2921.

Frank, A.S. and Z.S. Bruno, 2008. Embedded system security. J. Electron. Prod., 5: 111-113.

Guthaus, M.R., J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, 2001. MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the 4th IEEE Workshop on Workload Characterization, December 2, 2001, Austin, TX, USA., pp: 10-22.

Hu, W. and Z. Tang, 2003. Design of processor architecture of longxing 1. Chinese J. Comput., 26: 385-396.

Shao, Z., C. Xue, Q. Zhuge, E.H.M. Sha and B. Xiao, 2006. Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. IEEE Trans. Comput., 55: 443-453.

Wan, Y., Z. Liu and W. Cui, 2011. Study of a hardware defense mechanism to against buffer overflow attacks for embedded systems. Microelectron. Comput., 28: 135-137.

Zhang, S., 2008. Design and Application of Embedded System. Tsinghua University Press, Beijing, China, pp: 1-2.

Zhao, F., D. Ma and W. Sun, 2008. Design Examples of Embedded System Based on FPGA. Xi'an Electronic and Science University Press, Xi'an, China, pp: 4.