Hardware-Assisted System for Program Execution Security of SOC

Xiang WANG, Shu-Song PANG, Wei- Ke WANG, Zong- Min ZHAO, Cheng ZHOU, Zhan-Hong HE, Xiao-CUI ANG and Yang XU

School of Electronic and Information Engineering, Beihang University, Beijing 100191, China
[a] *Corresponding author: author@e-mail.org*

Abstract. With the rapid development of embedded systems, the systems' security has become more and more important. Most embedded systems are at the risk of series of software attacks, such as buffer overflow attack, Trojan virus. In addition, with the rapid growth in the number of embedded systems and wide application, followed embedded hardware attacks are also increasing. This paper presents a new hardware assisted security mechanism to protect the program's code and data, monitoring its normal execution. The mechanism mainly monitors three types of information: the start/end address of the program of basic blocks; the lightweight hash value in basic blocks and address of the next basic block. These parameters are extracted through additional tools running on PC. The information will be stored in the security module. During normal program execution, the security module is designed to compare the real-time state of program with the information in the security module. If abnormal, it will trigger the appropriate security response, suspend the program and jump to the specified location. The module has been tested and validated on the SOPC with OR1200 processor. The experimental analysis shows that the proposed mechanism can defence a wide range of common software and physical attacks with low performance penalties and minimal overheads.

1. Introduction

The Embedded systems have rapidly evolved over the last years, which are being widely applied to military affairs, economy and other areas of society. Its related products can be found everywhere in our daily life, including smart cards, mobile, sensors, wearable devices, vehicles, ET, al. Thus the security design has become the most significant issues in embedded systems.

Due to the wide number of embedded devices as well as the great progress of the network technology, the embedded systems can be easily attacked. When the operating system is running on the embedded system, the virus has the potential to attack embedded systems. Then control and tamper embedded devices. At the same time embedded system performance has been greatly improved, but due to its own resource constraints and environmental needs, making the software of PC and cannot be ported to embedded systems.

Attacks for embedded system generally have two ways: software attack and hardware attack. The software attack mainly contains buffer overflow, logic bombs, Trojan viruses and worms. The buffer overflow attack is considered most destructive. Its attacks include stack overflows, heap overflows, dangling pointer references, format string vulnerabilities and integer errors. Hardware attack usually includes irreversible attacks like die shear, chemical attacks and reversible attacks like error signal injection, voltage temperature variation. At this stage, software attacks are more common, and the hardware attack need the attacker have more expertise.

So far, software protection has been mainly adopted to defend against most attacks. For example, safe programming language source code or security analysis is often used to fix the vulnerability. However, software protection requires a lot of rewriting the code, which leads to significant performance loss. And security is also very limited. In this paper, we propose a hardware-assisted solution to ensure the safety of program execution by compiler offline extraction program control flow, static data and static code. During program execution, the security module monitors the control flow and verifies data and code integrity.

2. Related Work

The security protection methods of system on chip (SOC) have been rapidly increased over the past years. The software-based and hardware-based protections are showed as below.

Richard et al [1] proposed a code rewrite and in-lining system (REINS) method to protect the code from malicious software attacks of untrusted sources. It's a software-based approach to detect attacks. Recently Schuster et al [2] came up with a new type of attack, called counterfeit object-oriented programming (COOP).

It abuses the C++ virtual dispatch mechanism to reuses whole functions. Stephen [3] presented a software way which improve and simplify the COOP attack and use table randomization to resist Code-reuse attacks. Abadi et al [4] proposed a new method to monitor the control flow of program. It derives the program execution process by static analysis and then check with CFG in the runtime.

In recent years, hardware-based approaches became prevalent. Jean-Luc et al [5] adopted a simple hardware solution to check good execution of one program by checking that each basic block is correctly executed and that the Control Flow Graph (CFG) is respected. But their solution has some shortage in overhead and grain. Lucas [6] tackled these limitations and presented a security hardware mechanism to deal with fine-grained CFI checks. Arora [7] also presented a hardware assisted mechanism to check control flow and integrity of code. Bu [8] is similar to Arora's design which protects the code and data integrity, but no concern about control flow protection. Zhao [9] presents a code security mechanism, which can accurately positioning error type. Guo [10] proposed hardware-based intrusion detection way called Control-flow Verification System (CONVERSE), which guaranteed control-flow integrity by checking the destination of control-flow branches at runtime.

3. Security Architecture

This section describes the working principle and process of hardware assisted monitoring model.

3.1. Overview

Fig. 1 depicts the relationship between the monitoring model and CPU.

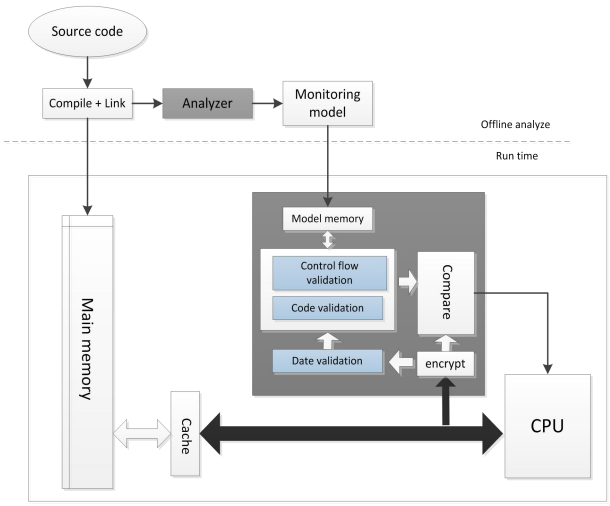

Figure1. The Monitoring Model Design Flow

3.2. The analyzer design

The main function of the analyzer is to analyze and extract static information from the object file. These include control flow information and code integrity information. The analyzer is written a script program running on PC. Here is a new definition of the concept of a basic block. Basic block refers to a code fragment which only contains the instructions executed sequentially. That is to say, there is no branching instruction in a basic block. The analyzer is marked by a jump instruction to divide the basic block, which means the starting and ending address of the basic block is the destination address of the jump instruction and the address of the next jump instruction. The designation of the analyzer is based on the basic block unit to extract and divide the program. The final result is used to initialize the security module.

Specific of extraction process are shown below:

Procedure1: Extraction process

P1: Variable initialization and some other settings, initialization operations. Enter the object file name. In addition, we also need to fill in some of the address information, such as address of boot code.

P2, P3: Executable file header contains a lot of information. The analyzer check the header files a) the file is a valid 32-bit executable ELF file or not. b) The platform this file run on is platform-specific or not. c) Extracting executable segment offset address.

P4: Call objdump tool to disassemble the object file.

P5: Functions of Interrupt table are end of jump instruction. But generally no program will call them directly. Therefore it cannot be inferred start address of basic block based on the jump instruction. So we want to add the start address based on specific documents.

P6: his is the most important step. After disassemble the executable file, actually obtained a text file of the disassembly format. By regular expressions, one can directly find the jump instruction and extract the address information based on the instruction format.

P7: Although P6 found the start and end addresses of all the basic blocks, but have not found the correspondence between the start and end address. This step would sort all addresses.

P8: After calculating the information of all the basic blocks, put each basic block instructions as input of lightweight hashing (L_Hash) algorithm and output a 16-bit result. It represents the completeness of code.

P9: Adding above information to the monitor model and use it to initialize Ram inside the security module.

3.3. The Code Integrity

This section describes the integrity protection program code, as well as the basic structure of the security model. Structure of monitoring model is shown in Table 1.

Table1. Monitoring Model of Code Integrity

Info	Wides	Address range
Current block start address	16bits	[59 :64]
Current block end address	16bits	[63:48]
L_Hash value between them	16bits	[47:32]
Successive basic block index S0	*16bits*	*[31:16]*

Info	Wides	Address range
Successive basic block index S1	*16bits*	*[15: 0]*

Current block start address:

The correct start address of basic block in currently executing program.

Current block end address:

The correct end address of basic block in currently executing program.

L Hash value between them:

The hash value of the currently executing program.

Next basic block start address:

The start address of next basic block.

After the security module is initialized, security module can start monitoring the execution of the program. At this time, the program counter of the CPU (PC) and instruction of decoding (ID) stage signal are connected to the security module. When the program starts running, the security module will monitor the execution of program. When the security module detects the start address of the program, L_Hash module becomes enabled. The current and subsequent instructions act as input of L_Hash function, until the end address of this basic block is detected by security module.

The next step is to calculate this basic block L_Hash value and compared with the L_Hash value stored in the security model. If not equal, the system must be attacked. Security model will output exception messages and freeze the CPU as well. If it do not detect a start or end address of basic block, it can be also concluded attack occurred and output the exception messages.

Procedure 2 describes how the security module protects the safety of the code.

Procedure 2 code integrity check flow

1: Inputs: PC(program counter) , Instruction
2: Output: abnormal_signal
3: B_S ← set of all basic blocks start address bb_si, 1 ≤ i≤
the number of basic block
4: B_E ← set of all basic blocks end address bb_ei
5: I ← set of instructions of basic block i insnij, 1≤ j ≤ the
number of instruction in a basic block
6: if pci = bb_si : bb_si ⊆ B_S then
7: L_Hashi = L_Hash(insni1,insni2.....insnij)
8: if pci = bb_ei : bb_ei ⊆ B_E then
9: if L_Hashi = monitoring model_i[16:31] then
10: abnormal_signal = 000 /*no error*/
11: else abnormal_signal = 001 /*l_hash value error*/
12: else abnormal_signal = 010 /*end address error*/
13: else abnormal_signal = 100 /*start address error*/

3.4.The Control Flow Validation

This section describes the procedures for integrity verification of the control flow. The control flow attack occurs mainly through tampering return address of function or changing the condition of jump. The attacks occur around the jump instruction. So the security module monitor addresses of jump instruction. For direct jump instruction, the jump address which can predict where an unconditional jump instruction has only one and conditional jump address had a maximum of two. For indirect jump instruction, since the jump address in a register or memory, address of jump destination cannot be predict offline, and therefore should be special treatment. The architectural to protect control flow use two 16-bit memories S0 and S1 to store target addresses as the shown in Table2. Generation of transition table is shown in Procedure 3.

Table2. Monitoring Model of Control Flow

Info	Wides	Address range
Current block start address	16bits	[59 :64]
Current block end address	16bits	[63:48]
L_Hash value between them	16bits	[47:32]
Successive basic block index S0	16bits	[31:16]
Successive basic block index S1	16bits	[15: 0]

Procedure 3 Transition Table generation

1: input: B E
2: B ← set of all basic blocks bi, 1 ≤ i ≤ the number of basic block
3: E ← set of direct edge eij ⊆ E ⇔ bj is a successor of bi
4:output:T={row0,row1,...,rown} where rowi is a tuple(index,s0,s1)
5: for all bi ⊆ B do
6: rowi.index = i
7: if bi end with indirect jump then
8: Targets ← bj : eij ⊆ E
9: $row_i.s_0$ = special code
10: rowi.s1 = ||Targets||
11: rowi.s0 = j : bj is the branch-not-taken target
12: rowi.s1 = k : bk is the branch-taken target
13: else
14: rowi.s0 = k
15: rowi.s1 = 0

3.5.The Data Integrity

Both data and code are necessary for the execution of program. Attack for data may not change the control flow or destroy the integrity of the instruction code in the execution of the program. Protecting data is more difficult than code. Code information can be known after compile and it would not change in the execution time. But data information is highly dynamic. When the program is running, only a small fraction of the data is fixed and most of the data is uninitialized or unallocated. The data integrity validation architecture is shown as Fig. 2.

For the static data which can be determined after compiled, it is extracted offline and stored in the static data monitoring model. For the dynamic data, the module starts to calculate the memory access address when checking the stored instruction. Then data pre-processing model calculates the L_Hash value of current data and stores in the Dynamic data monitoring model.

Similarly, the module will compare the data in the data monitor model of current address with the L_Hash value of data on the bus when checking the load instruction of external memory. If not equal, it will output an interrupt signal to ensure that data is not modified in the external transfer process.

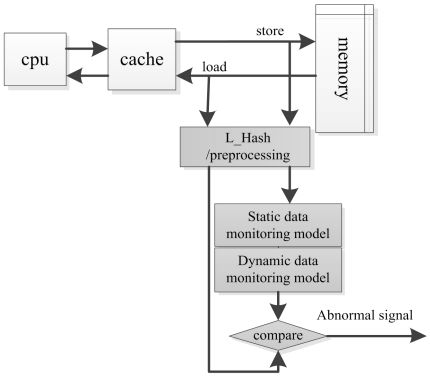

Figure2. The Data Integrity Validation Architecture

4. Analysis of Security

The design can defence many kinds of attacks. It's so difficult to do completely security analysis. So we divide attacks into the following categories.

4.1. Tampering Program Code

The design can defence most kinds of code attacks. The typical attack code as shown in the figure3.

```
C code of promgram:
int out=x+y; (normal)
int out=x-y; (tampered)

The disassembly code changes are as follows:
2118: e0 64 18 00    l.add r3,r4,r3 (normal)
2118: e0 64 18 02    l.sub r3,r4,r3  (tampered)
```

Figure3. The Tamper on Code

But there are two situations requiring attention. a) It cannot detect attacks immediately. The granularity of monitoring module is not every instruction but it was monitored using basic block as a unit. So it can detect the integrity of the code after a basic block executed in the program. This will decrease the detection efficiency of code attacks. But it can save the storage resource overhead and reduce the impact on the CPU's running speed. b) Collision attacks of checksum. L_Hash algorithm is applied in this design. It supports 80, 96, and 128 bits of three sizes of summary length and provides 64 bits to 120 bits between the original image of safety, 40 or 60 of the second preimage and collision resistance. Use serialization to achieve L_Hash, respectively, only 817 and 1028 gates. By adjusting the parameters, L_Hash can be a compromise among security, speed, energy consumption and the realization of the price and other indicators. At the same time, L_Hash has good performance in software implementation.

4.2.Tampering Control Flow

The detection of control flow is to monitor jump instructions. Common attacks are shown in Fig. 4. For direct jump which can be predicted its jump address offline, the design can be directly detect illegal jump address. But for the indirect jump instruction, the address of jump destination cannot be known before running the program. So directly detection of control flow is not available. However, considering the attack on the indirect jump instruction is either a direct attack on the code or the transfer of the conditions (data) of the attack. So it's also indirectly protected the control flow by protecting code and data.

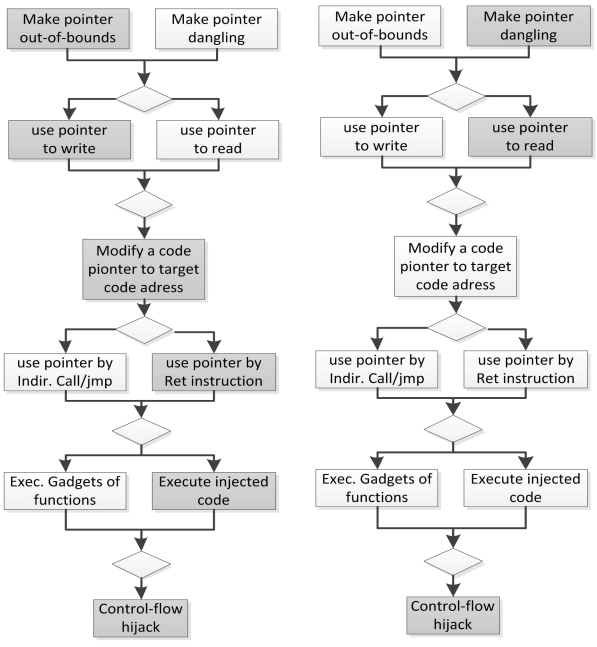

Figure4. The Tamper on Control Flow

4.3.Tampering Program Data

The attack of the data is mainly on the static data and dynamic data. Static data always exists in memory. The method of attack on that is to tamper memory content directly or attack in data transmission process. Dynamic

data exists in external memory or cache. The attack can occur in cache and the above two kinds of attack in static data. Due to the memory off chip, the data attacks are likely to occur in the process of data transmission. This design is mainly aimed at that type of attack. As shown in the Figure5.

The following special circumstances need to be explained. a) During the execution of program, data may not be accessed through the external data bus. The data may be in the cache. If the data in the cache is attacked, the security module will not detect it. b) If you directly change the data in the memory, then the attack can be detected, but it cannot be restored.

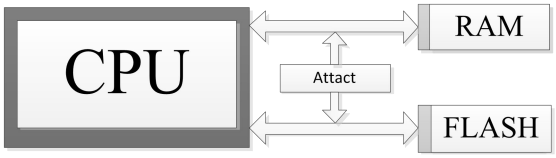

Figure5. The Tamper on Control Data

5. Experiment Result

In terms of platform building, OR1200 processor is adopted which is a 32-bit scalar RISC with Harvard micro architecture, 5 stage integer pipeline. The system on chip (SOC) platform is verified on a Xilinx virtex5 FPGA. Hardware resource consumption is shown in Table 3. The total used proportion of Slice Registers is about 25% and the Security module used used 3%. We can see this module has low resource consumption.

Table3. Platform Resource Consumption

Slice Logic Utilization	Used	Security module used	Available
Slice Registers	7,022	1,024	28,800
Slice LUTs	12,039	1,322	28,800
occupied Slices	4,716	585	7,200
LUT Flip Flop pairs used	14,185	1,672	
bonded IOBs	179	122	480
BlockRAM/FIFO	56	30	60
BUFG/BUFGCTRLs	8	2	32
Average Fanout of Non-Clock Nets	4.20	4.16	

6. Conclusion

This paper presents a hardware assisted for program secure execution mechanism. The proposed method relies on the compiler to generate the monitoring model at compile time; and the hardware monitor verifies the execution trace at runtime. The experimental result shows that the design can defend against code attack, data attack, and control flow attack effectivly. The structural design has run on an actual FPGA platform. Implementation of Implementation of security model has low resources consumption. And under normal circumstances, it meet the trade-offs between overhead and security.

Acknowledgment

This research is supported by the National Science Foundation of China (Grant No. 60973106, 81571142), the Key Project of National Science Foundation of China (Grant No. 61232009), National High-tech R&D Project of China (863 Grant No. 2011AA010404), and Fundamental Research Funds for the Central Universities, Astronautic Support Fund (Grant No.377054).

References

1. Wartell R, Mohan V, Hamlen K W, et al. Securing untrusted code via compiler-agnostic binary rewriting, Computer Security Applications Conference. 299-308 (2012)
2. Schuster F, Tendyck T, Liebchen C, et al. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications, Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, 745-762 (2015)
3. Crane S J, Volckaert S, Schuster F, et al. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 243-255 (2015)
4. Abadi, M.Budiu, U.Erlingsson, J.Ligatti. Control-Flow Integrity Principles, Implementations, and Applications, In ACM Conference on Computer and Communication Security (CCS) (Alexandria, VA), 340-353 (2005)
5. Danger J L, Guilley S, Porteboeuf T, et al. HCODE: Hardware-Enhanced Real-Time CFI, Proceedings of the 4th Program Protection and Reverse Engineering Workshop. ACM, **6** (2014)
6. Davi L, Koeberl P, Sadeghi A R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation, Proceedings of the 51st Annual Design Automation Conference. ACM, 2014: 1-6.
7. Arora Divya, Ravi Srivaths, Raghunathan Anand; Jha Niraj K, December 2006, Hardware-assisted runtime monitoring for secure program execution on embedded processors, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, **14**, 12 (2014)
8. Bu C, Wang X, Zhang C, et al. Compiler/hardware assisted application code and data security in embedded systems, Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th. IEEE, **7**. E. 2-1-7. E. 2-8 (2009)
9. Wang X, Zhao Z, et al. A Design of Security Module to Protect Program Execution in Embedded System, Green Computing and Communications (GreenCom),

IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing. IEEE, 1750-1755 (2013)

10. Guo Z, Bhakta R, Harris I G. Control-flow checking for intrusion detection via a real-time debug interface, Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on. IEEE, 87-92 (2014)