Proc. of The Fifth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-104-7 doi: 10.15224/ 978-1-63248-104-7-02

Leveraging Embedded Systems' Security by Hackers' Attack Detection

Bruno C. Porcher, Raphael S. Ferreira, Fabian Vargas, Letícia Bolzani Poehls, Ariel Lutenberg, Federico Zacchigna

Abstract - **In recent years, computer systems belonging to large companies, governments as well as personal computers have been experiencing an increasing wave of attacks that disrupt their normal operation or leak sensitive data. In this context, this paper presents a hardware-based approach which aims at detecting several types of attacks that degrade system security. The approach is based on a dedicated watchdog, which is tightly connected to the processor bus. Compared to existing approaches, the proposed technique can be applied to any application code "as it is", i.e., it does not need application code recompilation. Additionally, the approach does not use of any kind of supervisor software (e.g., an Operating System - OS) to manage memory usage. To validate the approach, a case-study based on the LEON3 softcore processor and security-vulnerable code snippets from benchmark test codes have been implemented. Experimental results indicate that this approach is able to detect an intrusion tentative for 100% of the test cases, while yielding low area overhead and negligible processor performance degradation. Moreover, attack detection latency depends on the user code complexity, which means that detection time can take from one up to two or three tens of machine clock cycles.**

Keywords - **Hacker's Attack, Malicious Code Injection, User Code Basic Block, Dynamic Integrity Checking, Secure Embedded System.**

I. Introduction

The need to include security mechanisms in electronic devices has dramatically grown with the widespread use of such devices in our daily life. In this scenario this work presents a hardware-based approach (here defined as a watchdog connected to the processor bus) which aims at leveraging electronic systems' security by detecting hackers' attacks. The detection is done at runtime, when the hash of the basic block of instructions under execution is different from the one statically computed, during the compilation time. In this case, the computed hash value is observed to be different because of a hacker's attack. The fundament behind the approach settles around the following actions: first, the user program is translated into a set of basic blocks. Second, the hash value for each basic block is computed by summing the opcode value of all instructions in the basic block. And third, the hash value is stored in a CAM (Content-Addressable Memory) used to directly map the first and last addresses of each basic block with its hash value. This CAM is instantiated as an internal block of a watchdog, which in turn is connected to the processor bus. At runtime, the watchdog monitors the instruction bus in order to identify basic blocks along with the user program currently under execution, compute their (dynamic) hash values and compare them against the hash values computed statically (at the compilation time). In this approach, every time the watchdog signals a mismatch comparison between both dynamic and static hashes, a potential attack is assumed to be detected.

Bruno C. Porcher, Raphael S. Ferreira, Fabian Vargas and Letícia Bolzani Poehls are with the Catholic University – PUCRS, Electrical Engineering Dept., Porto Alegre, Brazil
Ariel Lutenberg and Federico Zacchigna are with the University of Buenos Aires – UBA, Engineering Faculty. Buenos Aires, Argentina

The present paper represents an improvement of the work first presented in [1, 2] in the sense that it is able to detect attacks on indirect system calls (i.e., function calls implemented with pointers). Moreover, the proposed approach is able to efficiently detect DMA Attacks [3,4] and Hardware Trojan-Triggered Attacks [5] that result in user code hash changes.

II. Preliminaries

The primary goal of attackers is to strive to achieve victim's computer control. This goal is achieved by means of two mutually dependent steps:

i) *Inject attack code*: The attacker provides an input string that is actually an executable binary code native to the machine being attacked. Typically, this code is simple and does something similar to exec("sh") to produce a root shell.

ii) *Change the control-flow execution:* The control-flow execution is changed by executing the injected input string, which actually points to the attack code.

The proposed approach is able to detect the most important types of attacks that degrade system's security. These attack types are described hereafter:

1) *Stack Smashing Buffer Overflow Attack:*

Buffer overflow attacks [1,2,6] exploit a lack of bounds checking on the size of input being stored in a buffer array in memory. By writing data *past* the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. By far, the most common data structure to corrupt in this fashion is the stack, called a "stack smashing" or "buffer overflow" attack.

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible.

Proc. of The Fifth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-104-7 doi: 10.15224/ 978-1-63248-104-7-02

2) *DMA Attack:*

DMA is included in a number of connections, because it lets a connected device (such as a camcorder, network card, storage device or other useful accessory or internal PC card) transfer data between itself and the computer at the maximum speed possible, by using direct hardware access to read or write directly to main memory without any operating system (OS) supervision or interaction. The legitimate uses of such devices have led to wide adoption of DMA accessories and connections, and an attacker can equally use the same facility to create an accessory that will connect using the same port, and can then potentially gain direct access to part or all of the physical memory address space of the computer, bypassing all OS security mechanisms and any lock screen, to read all that the computer is doing, steal data or *cryptographic keys*, install or run *spyware* and other *exploits*, or modify the system to allow *backdoors* or other malware [3,4].

Preventing physical connections to such ports will prevent DMA attacks. On many computers, the connections implementing DMA can also be disabled within the BIOS or UEFI (Unified Extensible Firmware Interface) if unused, which depending on the device can nullify or reduce the potential for this type of exploit. Examples of connections that may allow DMA in some exploitable form include FireWire, CardBus, ExpressCard, Thunderbolt, PCI, and PCI Express.

3) *Hardware Trojan-Triggered Attack:*

An integrated circuit (IC) could provide satisfactory functionality for its designed specification, but also may contain malicious logic (i.e., Hardware Trojan: HT). HT can be embedded into the circuit such that it remains asleep until activated. Hardware Trojans can be naive, very simple modifications to the original circuit [5]. In general, HTs try to bypass or destroy major security concerns of any system by: leaking confidential information and secret keys covertly to the adversary (Confidentiality attack); changing the value of a certain register (Integrity attack); disabling, deranging or destroying the entire hardware or components of it (Availability attack). Traditional hardware testing strategies cannot effectively detect HTs because the probability of triggering HT during functional testing is extremely low.

Related Works:

Several efficient software-based as well as hardware-based dynamic integrity checking techniques [7,8] have been proposed in the literature. However, software-based techniques suffer from performance overheads as high as 60%, while hardware-based approaches result in average overheads of about 18% [9]. These are daunting numbers. Additionally, some of these approaches [9,10] need application code recompilation to compute specific information (hashes of application program's instruction addresses and opcodes) that is later used at runtime to detect attacks. And finally, from the best of our knowledge, we could not find in the literature any work able to jointly detect the three above mentioned attack types.

III. The Proposed Approach

The proposed approach is based on two specific structures (a) The implementation of a watchdog in hardware and (b) on the reservation of a dedicated CAM (Content-Addressable Memory) which sits beside the watchdog. In more detail, the approach works as follows (see Fig. 1a):

- First, the user program is translated into a set of basic blocks. A basic block is a set of code instructions that is sequentially executed by the processor, without any conditional or unconditional branch. So a basic block is the existing code snippet between two consecutive branch instructions. A basic block always finishes by a branch instruction.
- Second, the static hash value for each basic block is computed (at the compilation time) by summing (XOR operating) the opcode value of all instructions in the basic block.
- Third, the static hash value is stored in a CAM used to directly map the first address of each basic block with its hash value and the number of instructions composing this code snippet (Fig. 1b). This CAM is instantiated as an internal block of a watchdog, which in turn is connected to the processor bus.
- Fourth, at runtime, the watchdog monitors the instruction bus in order to identify basic blocks along with the user program currently under execution, compute their (dynamic) hash values and compare them against the static hash values. Every time the watchdog signals a mismatch comparison between both dynamic and static hashes, this means that the current sequence of instructions under execution is not the one expected and thus, a potential attack is assumed to be detected.

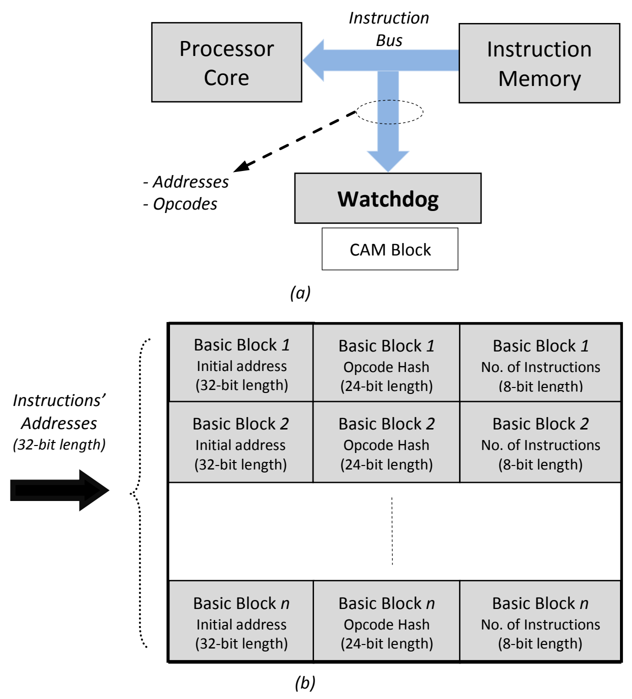
Instruction Bus
Processor Core
Instruction Memory
- Addresses
- Opcodes
Watchdog
CAM Block
(a)
Instructions' Addresses (32-bit length)
Basic Block 1 Initial address (32-bit length)
Basic Block 1 Opcode Hash (24-bit length)
Basic Block 1 No. of Instructions (8-bit length)
Basic Block 2 Initial address (32-bit length)
Basic Block 2 Opcode Hash (24-bit length)
Basic Block 2 No. of Instructions (8-bit length)
Basic Block n Initial address (32-bit length)
Basic Block n Opcode Hash (24-bit length)
Basic Block n No. of Instructions (8-bit length)
(b)

Fig. 1. General architecture of the proposed approach: *(a)* Block overview; *(b)* CAM memory structure.

Proc. of The Fifth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-104-7 doi: 10.15224/ 978-1-63248-104-7-02

Fig. 2 summarizes the internal blocks of the watchdog. As depicted, the watchdog is tightly connected to the processor pipeline by one side and to the instruction memory bus by the other side.

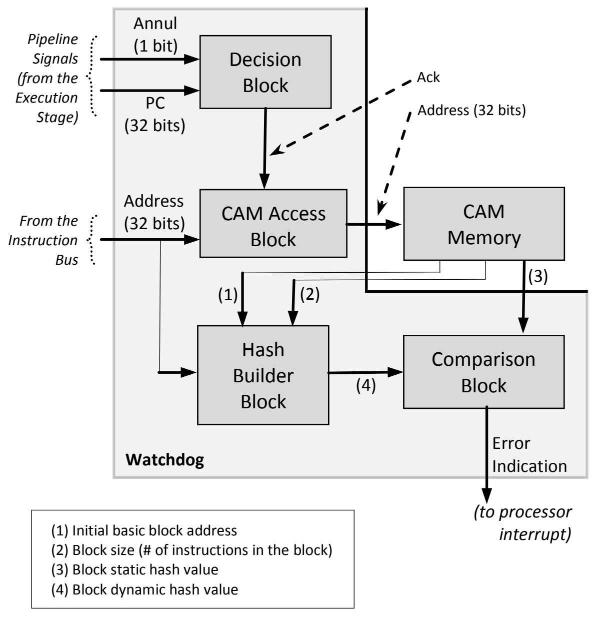

Fig. 2. Internals of the Watchdog.

Based on Fig. 2, the watchdog operation is as follows:

1) The watchdog monitors some internal signals from the execution stage of the processor pipeline:

- the bit "*annul*", whose function is to indicate if the instruction that is leaving the Execution Stage of the pipeline will actually be executed by the processor or it will be discarded due to speculative execution.

- the "*Program Counter*" (PC), which is saved into the Decision Block and is further used to identify the 1st instruction of basic blocks.

2) If the current PC is actually executed in the pipeline, then the Decision Block acknowledges positively to the CAM Access Block, which searches in the CAM if the executed PC corresponds to the 1st instruction of a basic block. If so, the CAM memory forwards:

- this instruction address and the number of instructions composing the basic block (computed previously by code static analysis) to the Hash Builder Block, and
- the static hash value of such basic block to the Comparison Block.

3) In the sequence, the Hash Builder Block computes, instruction by instruction, the dynamic hash value for the block under execution till the CPU reaches the last instruction of the basic block. Then, this dynamic value is sent to the Comparison Block.

4) The Comparison Block compares the static hash value against the dynamic one, computed by the Hash Builder Block. If this comparison is true, no decision is taken; otherwise, an error indication is sent to the processor, which is used to interrupt code execution.

From the above described, we can extract two conclusions:

1) The size of the watchdog is constant and quite small since the complexity of the computation to be performed by the watchdog is relatively low: series of XOR logical operations in order to: (a) generate (dynamicaly) instructions' hashes, (b) compare the static hash against the dynamic one at the end of a basic block, (c) detect the execution of the first instruction of a basic block, (d) count the number of instructions contained in a basic block and ultimately (e) access the CAM memory.

2) The size of the CAM memory is variable; it depends on the user code complexity. In more detail, it depends on the number of basic blocks the code is devided into. Every first instruction address of each basic block is a CAM entry, which points to the block hash value and number of instructions contained in the basic block (Fig. 1b).

IV. Experimental Results

This approach was implemented on the LEON3 softcore processor [11]. The LEON3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The model is highly configurable, and particularly suitable for system-on-a-chip (SoC) designs. Fig. 3 depicts the general block diagram of the processor core. Blocks indicated by (*) are optional and are included in the processor main architecture if selected by the designer. Therefore, the basic processor configuration is the LEON3 CPU Integer Unit, the AMBA AHB Master Interface and the AMBA Bus, which connects the CPU to the system memory.

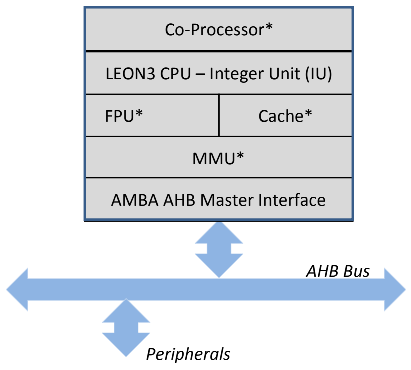

Fig. 3. General block diagram of the LEON3 softcore processor.

With the purpose of validating the proposed approach, test programs were implemented with pieces of known vulnerable C codes (see Table 1). These vulnerable pieces of C code were obtained from vulnerable test benchmarks published in the CVE (Common Vulnerabilities and Exposures) [12,13]. These code snippets were adapted and included into the test program source codes. Then, while running these programs the watchdog was evaluated. For 100% the code snippets' executions, the watchdog was able to detect the changes

Proc. of The Fifth Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2016
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-104-7 doi: 10.15224/ 978-1-63248-104-7-02

intentionally injected in the code, such as changes in the return addresses from system function calls.

Vulnerable Programs	CVE Number	Severity
Edbrowse	CVE-2006-6909	10.0 high
MADWiFi	CVE-2006-6332	7.5 high
OpenSER	CVE-2006-6749	9.3 high
Samba	CVE-2007-0453	4.6 medium
Sendmail	CVE-2003-0681	7.5 high
Wu-ftpd	CVE-1999-0368	10.0 high
Wu-ftpd	CVE-2003-0466	10.0 high

Table 1. Benchmark vulnerable code snippets [12,13].

Table 2 shows the area overhead added by the watchdog implementation with respect to the LEON3 processor. The obtained numbers were computed for the couple (processor + watchdog) synthetized by the PlanAhead CAD tool for the Xilinx Spartan3E FPGA. It is worth noting that these numbers include the watchdog main body as depicted in Fig. 1a, leaving aside the CAM memory, whose number of entries is variable and depends on the user code complexity (i.e., it depends on the number of basic blocks in the code). This table depicts the resulting area overhead for the main FPGA infrastructure components (number of extra flip-flops: FLOP_LATCH, look-up tables: LUT, multiplexers: MUXFX, I/O pins and SRAM cells: DMEM). As observed, the average area overhead after addition of the watchdog is 2.16% of the original LEON3 processor area.

Primitive Type	Leon + Watchdog	Watchdog Entity	Area Overhead [%]
FLOP_LATCH	2649	135	5.1%
LUT	8026	109	1.36%
MUXFX	663	4	0.6%
IO	117	0	0%
DMEM	13	0	0%
Total	**11,468**	**248**	**2.16%**

Table 2. Area overhead yielded by the Watchdog implementation.

V. Final Considerations

This paper presented a hardware-based approach to protect systems from stack smashing attacks.

Experimental results show that this approach successfully detected 100% of the injected attacks under the analyzed situations. The approach yields low area overhead (2.61% for the case studied: LEON softcore processor mapped into a Xilinx Spartan3E FPGA. Moreover, no performance degradation was observed since the watchdog operates in parallel with processor execution. Finally, attack detection latency depends on the user code complexity, which means that detection time can take from one up to two or three tens of machine clock cycles.

Currently, we are implementing a detailed case-study in order to compute the attack detection latency of the watchdog. Such latency, in more detail, depends on the distance the malware point is from the end of the basic block. In this context, by "malware point", we understand the point where the user code is intentionally changed (by adding/removing/changing instructions) to damage or disable computers and computer systems.

Acknowledgment

This work has been supported in part by CNPq (National Science Foundation, Brazil) under contract n. 306619/2015-6 (PQ) and Hewlett-Packard Brazil Ltd. using incentives of Brazilian Informatics Law (8.2.48 from 1991).

References

[1] R. Segabinazzi Ferreira, F. Vargas. "ShadowStack: A new approach for secure program execution", Microelectronics and Reliability Journal, 55(9) August 2015, pp. 2077-2081.

[2] R. Segabinazzi Ferreira, F. Vargas, L. Bolzani Poehls. "Hardware-Based Stack Smashing Attack Detection & Preliminaries on Recovery Procedure", 4th International Conference On Advances in Computing, Control and Networking - ACCN 2016, Bangkok, Thailand, May 07-08, 2016, pp. 17-20 (DOI: 10.15224/978-1-63248-095-8-05).

[3] M. Dornseif, "0wned by an iPod", Proceedings of the 2nd PacSec Applied Security Conference, November 2004.

[4] A. K. Kanuparthi, R. Karri, G. Ormazabal, S. K. Addepalli, "A High-Performance, Low-Overhead Microarchitecture for Secure Program Execution", 2012 IEEE 30th International Conference on Computer Design, ICCD 2012, Montreal, Canada, 9/30/12-10/03/12, pp. 102-107.

[5] A. Aliyu, A. Bello, U. Joda Mohammed, I. Hussaini Alhassan. "Hardware Trojan Model For Attack And Detection Techniques", International Journal of Scientific & Technology Research, Vol. 3, Issue 3, March 2014, pp. 102-105.

[6] B. P. Miller, D. Koski, C. Pheow Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl. "Fuzz Revisited: A Reexamination of the Reliability of UNIX Utilities and Services", Report: University of Wisconsin, 1995.

[7] M. L. Corliss, E. C. Lewis, A. Roth, "Using DISE to Protect Return Addresses from Attack", Workshop on Architectural Support for Security and Anti-Virus (WASSA), Oct. 2004.

[8] Y. Park., Z. Zhang, G. Lee, "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks", IEEE Micro, vol. 26 , issue 4, July-Aug. 2006.

[9] A. K. Kanuparthi, R. Karri, G. Ormazabal, S. Addepalli, "A High-Performance, Low-Overhead Microarchitecture for Secure Program Execution", IEEE International Conference on Computer Design (ICCD), Oct 2012, Montreal, Canada.

[10] M. A. Schuette, J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams", IEEE Transactions on Computers, vol. 36, no. 3, March 1987, pp. 264-276.

[11] URL: http://gaisler.com/index.php/products/processors/leon3. Last visit: July 2016.

[12] Common Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names. URL: https://cve.mitre.org/. Last access: July 2016.

[13] National Vulnerability Database - vulnerability search. URL: https://web.nvd.nist.gov/view/vuln/search. Last access: July 2016.