Hardware-assisted integrity monitor based on lightweight hash function

Pei Du^{a)}, Xiang Wang, Weike Wang, Lin Li, Tongsheng Xia,
and Hongge Li

*School of Electronic Information Engineering, Beihang University,
Beijing 100191, China*

a) ziyoumudi@buaa.edu.cn

Abstract: Security of embedded system is widely noticed for its common usage and open application environment. A hardware-assisted monitoring architecture based on lightweight hash function is proposed to detect run-time program integrity on the embedded processor. The fine-grained property information is extracted as the integrity verification object, and hashed by lightweight hash function as the monitoring model. The hardware architecture is implemented on an SoPC platform. Take five standard benchmarks for experimental objects, the experiments show that the proposed monitor accounts for less than 8.37% area overheads of the processor, and the average CPI of our secure processor with pipelined lightweight hash functions increases no more than 6.36%.

Keywords: embedded system security, integrity monitoring, lightweight hash function, property information

Classification: Integrated circuits

References

- [1] M. Rahmatian, *et al.*: “Hardware-assisted detection of malicious software in embedded systems,” *IEEE Embed. Syst. Lett.* **4** (2012) 94 (DOI: [10.1109/LES.2012.2218630](https://doi.org/10.1109/LES.2012.2218630)).
- [2] D. Arora, *et al.*: “Architectural support for run-time validation of program data properties,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **15** (2007) 546 (DOI: [10.1109/TVLSI.2007.896913](https://doi.org/10.1109/TVLSI.2007.896913)).
- [3] K. Patel, *et al.*: “Ensuring secure program execution in multiprocessor embedded systems: A case study,” *International Conference on Hardware/Software Codesign and System Synthesis* (2007) 57 (DOI: [10.1145/1289816.1289833](https://doi.org/10.1145/1289816.1289833)).
- [4] R. G. Ragel and S. Parameswaran: “IMPRES: Integrated monitoring for processor reliability and security,” *ACM Design Automation Conference* (2006) 502 (DOI: [10.1145/1146909.1147041](https://doi.org/10.1145/1146909.1147041)).
- [5] S. Lemey, *et al.*: “Wearable flexible lightweight modular RFID tag with integrated energy harvester,” *IEEE Trans. Microw. Theory Techn.* **64** (2016) 2304 (DOI: [10.1109/TMTT.2016.2573274](https://doi.org/10.1109/TMTT.2016.2573274)).
- [6] X. Wang, *et al.*: “Hardware-assisted monitoring for code security in embedded system,” *IEEE 15th Intl. Conf. Scalable Computing and Communications* (2015) 1393 (DOI: [10.1109/UIC-ATC-ScalCom-CBDCCom-IoP.2015.251](https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCCom-IoP.2015.251)).

- [7] J. Muramatsu and S. Miyake: “Hash property and fixed-rate universal coding theorems,” *IEEE Trans. Inf. Theory* **56** (2010) 2688 (DOI: [10.1109/TIT.2010.2046214](https://doi.org/10.1109/TIT.2010.2046214)).
- [8] A. Bogdanov, *et al.*: “Hash functions and RFID tags: Mind the gap,” *Cryptographic Hardware and Embedded Systems* (2008) 283 (DOI: [10.1007/978-3-540-85053-3_18](https://doi.org/10.1007/978-3-540-85053-3_18)).
- [9] A. Bogdanov and M. Knezeniv: “SPONGENT: The design space of lightweight cryptographic hashing,” *IEEE Trans. Comput.* **62** (2013) 2041 (DOI: [10.1109/TC.2012.196](https://doi.org/10.1109/TC.2012.196)).
- [10] J. Guo, *et al.*: “The PHOTON family of lightweight hash functions,” *Advances in Cryptology - CRYPTO* (2011) 222 (DOI: [10.1007/978-3-642-22792-9_13](https://doi.org/10.1007/978-3-642-22792-9_13)).
- [11] W. Wu, *et al.*: “LHash: A lightweight hash function,” *Conference on Information Security and Cryptology* (2013) 291 (DOI: [10.1007/978-3-319-12087-4_19](https://doi.org/10.1007/978-3-319-12087-4_19)).
- [12] C. De Cannière, *et al.*: “KATAN and KTANTAN-A family of small and efficient hardware-oriented block ciphers,” *Cryptographic Hardware and Embedded Systems* (2009) 272 (DOI: [10.1007/978-3-642-04138-9_20](https://doi.org/10.1007/978-3-642-04138-9_20)).

1 Introduction

With the development of information technology and microelectronic technology, demands for embedded systems are growing in such fields as communication, automation, network, etc. The embedded system brings us convenience, but also security threats. As the hardware performance and software complexity of the embedded system upgrade, software vulnerabilities are increasing accordingly. Moreover, the buffer overflow attacks are considered to be the biggest threat in software attacks [1] as shown in Fig. 1. The corresponding measures can be broadly divided into three classes: static check, real-time software check, and hardware-assisted check. The mechanisms of the first two methods are similar to the traditional computer protection. The last method is aimed at the characteristics of the embedded system, and thus its efficiency is usually higher than the first two methods, and the coverage of attacks is wider. With these superiorities, hardware-assisted check has attracted more and more attention, and several representative references are as follows.

Fig. 1. Breakdown of vulnerabilities reported by CERT for January–July (2017).

Arora proposes a hardware assisted run-time monitoring architecture by detecting and preventing unintended program behaviors [2]. The properties of the program at three levels of granularity are extracted by static program analysis,

and the extracted properties are compared with the program behaviors in real-time. A slight deviation of processor's dynamic execution trace will trigger designated response mechanism.

Patel exploits timing to ensure secure program execution in multiprocessor embedded systems [3]. The program is divided into many basic blocks at first, and then the operation time of each block is calculated and inserted into the head and tail of the basic blocks. When the program executes, secure hardware will check these operation times to ensure system security.

Ragel designs a novel scheme, Integrated Monitoring for Processor Reliability and Security (IMPRES) [4]. It is similar to Arora's monitoring architecture, but the hash values extracted in the static phase are directly inserted into program. The results show that the hardware area requirements of this scheme increase by 0.91%, the average clock cycle of each code increases by 11.9%, and the code volume increases by 10.6%.

Traditional hash functions, such as SHA-1(5527 GE), SHA-2(10868 GE) and MD5(8400 GE), are too large to fit for resource-constrained devices and be exploited in above researches and most hardware-assisted monitor architectures. The embedded system has strict resource constraints in terms of computing power, memory, and energy, and reserves no more than 20% hardware resources prepared for security. As a matter of fact, 64-bit or 80-bit preimage security is often considered as an appropriate secure value in embedded applications [5].

A secure monitor architecture based on lightweight hash function is proposed in [6]. A lightweight hash function LHash is used as the integrity algorithm to hash the extraction information in the static phase and run-time. The monitor uses the jump addresses as the monitoring information. Due to the overlapping basic block the monitoring information contains not only the start-stop addresses of the current instruction, but also the first possible jump address and the second possible jump address of the basic block, and costs 96 bits for each basic model. In that case, the monitor may ignore the instruction tampering with the correct address.

Different from [6], we utilize the start-stop addresses and the instructions of each basic block to generate the monitoring model. It effectively prevents the instruction tampering. And three different lightweight hash functions are applied in the monitor. After a comparison of the three lightweight hash functions, we improve a few components of the functions to further reduce the hardware implementation of the secure monitor. Then we complete functional simulation and synthesis of the monitor, and implement the proposed monitor hardware architecture on an OR (Open Reduce Instruction Set Computer, OpenRISC) 1200-FPGA platform. At last, some comparisons with the related works are given.

2 Hardware-assisted integrity monitoring scheme

In this section, we provide an overview of the proposed hardware-assisted monitor, and the steps to extract property information in static analysis. Since the monitor supervises the execution of the embedded processor, its self-protection is fatal. In our scheme, the connection between the processor and monitor is supposed to be secure, and the monitor is tamper-evident. Therefore, it is impossible for an attacker

to destroy the monitor without knowledge of the legitimate system host. Besides, other system components, such as I/O devices, external memory, buses connecting and on-chip caches are assumed to be insecure. And the program running in static analysis time is assumed to be safe.

2.1 Architecture overview

Fig. 2 shows the conceptual block diagram of the proposed hardware-assisted monitoring architecture. Most software attacks, including buffer overflow, can be detected by overseeing instructions of basic blocks and jump addresses of the program. Accordingly, we utilize these two kinds of information as property information for monitoring violation of permissible program behaviors as follows.

Fig. 2. Architecture and details of the proposed hardware-assisted run-time monitor

- At the compile time, property information is distilled from the source code of the program through the extraction tool which is introduced in detail at Section 2.2.
- The property information is used to generate the digest information block which consists of 16-bit start address, 16-bit stop address, 16-bit hash value of instructions, and 16-bit '0' reserving for higher security requirements.
- As the static integrity monitoring model, the digest information blocks of the whole permissible program are stored in the on-chip RAM of FPGA.
- At the run time, the monitor receives the executive flow produced by the processor, and hashes the real-time executive information. At last the monitor compares the hash values of the static integrity monitoring model and the run-time executive flow to generate control signals.

For ease of illustration, the embedded OR1200 processor is described as an in-order five-stage pipeline as shown in Fig. 2. The inputs to the monitor include the program counter (PC) and instruction register (IR) of the completing instruction. The monitoring model is divided into two parts: start-stop addresses stored in RAM.ad, and hash values of the basic block instructions in RAM.bb. In fact, the monitor searches the corresponding static monitoring model from RAM.bb by PC value. Apart from this, the monitor's outputs include a frozen signal and an alert signal. Once the monitor detects a possible violation behavior, it asserts the alert signal to generate a highest-priority interrupt to the processor. Moreover, this signal will trigger an interrupt response, for instance, terminating the program and switching the processor to a secure mode. The frozen signal is asserted when the monitor loses the pace of processor execution. This signal is treated as the normal state, while all pipeline stages are frozen till the monitor catches up with the processor.

2.2 Property information

Almost all software attacks are reflected in the fine-grained information, such as instruction and address. The basic block is referred to as an elementary unit of the control flow diagram. Therefore, instructions of basic block and its start-stop jump address are taken as the fine-grained property information in our monitoring scenario to guarantee integrity of the executive program. The monitoring model is generated at the compile time, namely, during the static analysis period. The basic block is a sequence of instructions executed orderly, and includes jump instructions which reflect all possible execution paths in the program. Except for the last jump instruction and first entry address, each block does not contain jump branch instructions. The detection mechanism includes both instruction and address, so that it is theoretically viable to resist most software attacks, especially buffer overflow.

Fig. 3. Diagram of extracting property information in permissible program

There are three possible entry statements of the basic block as follows.

- The first instruction;

- The statement that can be transferred by a conditional or unconditional transfer statement;
- The statement that follows the conditional transfer statement (in consideration of delay slot, we adopt the second statement following the conditional transfer statement).

After an analysis of the entire permissible program through above extraction tool in Fig. 3, it is convenient to determine the entry statements of each basic block, and complete the partition of the basic block. We set up a work environment on Ubuntu, and exploit a makefile to complete source code selection, compilation, target program analysis and program initiation.

3 Lightweight hash function

The hash function is a function that maps a variable length input to a fixed length hash value (or digest value), and widely used as the integrity verification algorithm [7]. Traditional hash functions based on the Merkle-Damgard construction are too large to fit for the embedded system. This is a new trend that the lightweight hash function is designed by using sponge construction as the external integral structure and the lightweight cipher as the internal permutation. The sponge construction has better security than the Merkle-Damgard construction in respect of resisting long message attack and random language machine discrimination attack [8].

3.1 Structure of lightweight hash functions

The sponge construction is a kind of compressed structure. Given a variable length input, the structure can produce an output with an arbitrary bit width. The steps of the sponge construction are concluded in three phases as depicted in Fig. 4 [9].

Fig. 4. Sponge structure inspired by different internal permutations

- **Initialization:** The message M is padded by 1-bit '1' and a reckonable number of '0' bits up to a multiple of r bits. After that, the padded message is split into m_i of r bits.

- **Absorbing:** Secondly, the initial state S_0 with b -bit is updated by xoring r -bit m_0 and permuted by function F round by round until all r -bit m_i are disposed.
- **Squeezing:** the first r' bits of the state are returned as the output, and the state is then updated by permutation F until n bits are returned as the entire output.

Specifically, the size of state b ($b = r + c > n$) is called width, where r is the rate and c is defined as the capacity. We refer to various parameterizations of hash functions as XXX- $n/c/r$. SPONGENT [9], PHOTON [10] and LHash [11] are opted as the integrity verification algorithm in our secure monitor architecture. Their structures are detailed in Fig. 4, which shows the three functions entirely employ sponge construction as external structure, and inspired by different internal permutations. SPONGENT uses PRESENT-like structure, while PHOTON employs AES-like framework, and the inner permutation of LHash is Feistel structure.

In aspect of security, the hash function has three properties as follows.

- **Collision resistance:** It is computationally impossible to seek out any two different inputs that produce the same output, such as $h(M) = h(M')$.
- **Preimage resistance:** If there is a designated message M , it is simple to compute the output $h(M)$. But for a given $h(M)$, it is computationally impossible to find out the input message.
- **Second-Preimage resistance:** If there is a specific message, it is hard to find another message that has the same hash value as the specific one.

The internal iterative permutation is assumed to be a stochastic function, and then the security properties of the hash function based on the strict sponge construction can be concluded as Eq. (1).

$$\begin{cases} \text{Collision: } \min\{2^{n/2}, 2^{c/2}\} \\ \text{Second-preimage: } \min\{2^n, 2^{c/2}\} \\ \text{Preimage: } \min\{2^n, 2^c, \max\{2^{n-r}, 2^{c/2}\}\} \end{cases} \quad (1)$$

3.2 Hardware implementation

Taking hardware cost and efficiency into account, six lighter parameterizations of the three lightweight hash functions are selected as the integrity verification algorithm in the monitor. Their security boundaries are shown in Table I.

Table I. Serial hardware implementation and security boundaries of lightweight hash functions in 180 nm process

Hash function	Block size (bit)	Area (GE)	Security (bit)		
			Pre.	2 nd Pre.	Coll.
Lhash-1(80/80/8)	88	784	72	40	40
Lhash-2(128/120/8)	128	955	96	60	60
SPONGENT-1(88/80/8)	88	759	80	40	40
SPONGENT-2(128/128/8)	136	1103	120	64	64
PHOTON-1(80/80/20)	100	856	64	40	40
PHOTON-2(128/128/16)	144	1394	112	64	64

In consideration of the area, hardware implementations are all in serial. The serial hardware implementations of SPONGENT and PHOTON are detailed in [9, 10]. Fig. 5 describes the permutation hardware implementations of LHash. The

control logic of LHash includes an arithmetic counter for round counting and some added combinatory logic to control the multiplexers. To reduce the cost, similar to KATAN [12] we replace the arithmetic round counter by a shift register-based counter, which saves 21 GE in 180 nm process. And the scan flip-flops is employed as an association of a D flip-flop and a dual-input multiplexer. Three zero-input multiplexers Mul_1 , Mul_2 and Mul_3 are used to initialize all flip-flops, and the round constant is referred to as g_i . Message block is separated as m_1 and m_2 , and m_1 represents as the left half bits of message block, and m_2 as the right half bits. The $P_{96/128}$ module can be regarded as a linear layer, and costs no extra logic except some wiring. Through hardware optimization the hardware implementations of LHash-1 and LHash-2 in 180 nm process cost 784 GE and 955 GE, and totally save 48 GE and 76 GE.

Fig. 5. Serialized hardware architecture of LHash

The three lightweight hash functions are synthesized by Synopsys Design Compiler (Version: D-2010.03-SP4 with the virtual silicon standard cell library UMCL18G212T3). For there is only one S-box in each round, the data path is designated as 4. Security and area of the six instances are listed in Table I. It can be seen that with similar parameters the three lightweight hash functions can provide 72–120 bits of preimage security, 40–64 bits second-preimage security and 40–64 bits collision resistance, satisfying 64-bit preimage security of embedded security requirement. Moreover, hardware implementation of the lightest hash functions needs only 759 GE, while the heaviest hash function costs only 1394 GE, far less than the area of SHA-1 (5527 GE).

4 Experiment and analysis

After the functional simulation, we build a platform for realizing run-time secure program execution on the embedded processor by monitoring property information integrity of the permissible program.

4.1 Simulation and resource overheads

Experimental SoPC platform is the OR1200 (svn rev 853) processor built on Digilent Genesys Virtex-5 FPGA (XC5VLX50T), in which “svn” refers to the SVN library provided by OpenCores.org. Environment of simulation and synthesis is Xilinx ISE 14.6. The extraction tool consists of or32-elf-gcc (4.5.1-or32-1.0rc4), or-elf-objdump (2.20.1.20100303) and or32-elf-objcopy (2.20.1.20100303).

In Fig. 6 the signal “flag_error” gives the result compared by monitor at the last instruction of the basic block. In that way, the monitor indicates whether the processor is attacked. The signal “id_pc_i” represents the address, while “f0800000~f0800028” includes just a whole basic block. In addition, “id_insn_i” means the instruction of the according address in basic block.

Fig. 6. Functional simulation of hardware-assisted run-time monitor with SPONGENT-88/80/8 for secure program execution. The red wave is the monitoring model, while the blue wave is the run-time monitoring model.

Fig. 6(a) is the normal running state when the run-time information hash value matches the monitoring model at the last instruction of the basic block. The “flag_error” is set to be “0”. Nevertheless, Fig. 6(b) shows the abnormal situation while we inject man-made revisions in that basic block. Therefore, the hash value of run-time information does not match the static monitoring model. Meanwhile, the signal “flag_error” is set to be “1”. For convenient observation and record, alert operation is temporarily neglected.

For estimating the area overheads of our monitor in the secure processor, we use DC (Synopsys Design Compiler) D-2010.03-SP4 with the VST (Virtual Silicon) standard cell library UMCL18G212T3 as synthesis environment. And Fig. 7 shows the area overheads proportions of different monitors in their processors. The purple blocks represent our work, and the blue blocks are the related researches. In addition, Arora’s monitor [2] is based on an ARM920T 32-bit processor core and synthesized with DC CB-130M COMS standard cell library. Patel’s monitor [3] uses Xtensa LX processor. And Ragel [4] exploits TSMC’s 90 nm core library for synthesis with typical conditions enabled.

Fig. 7. Area overheads for the monitor in the entire secure processor.

Fig. 7 shows the area overheads of the monitor in the processor. MSP-1 represents the proposed monitor with SPONGENT-1(88/80/8), MPH-1 is the proposed monitor with PHOTON-1(80/80/20), and MLH-1 means the proposed monitor with LHash-1(80/80/8). The hash engines are all pipelined, and the length of hash value is 16-bit. Adpcm, g721encode, g721decode, mpeg2encode, and mpeg2decode are the industry standard application benchmarks. Due to replacing the traditional hash functions by lightweight hash functions and exploiting the fine-grained property information, our monitor generally takes less than the Arora's monitor [2] which uses three levels of granularity for detecting. Moreover, Patel's [3] and Ragel's monitors [4] account for 10.7% area and 10.6% area overheads of their processors, all greater than the heaviest monitor (8.37%) proposed in this paper.

Besides, we evaluated the performance impact of the proposed architecture using the SimpleScalar 3.0/PISA architectural simulation tool set. Fig. 8 plots the average cycles per instruction (CPI) for several benchmarks with unpipelined and pipelined hash engines. As expected, with unpipelined hash engines, the performance penalty for the secure processor is mainly attributed to the latency of hash engine. Moreover, in Arora's monitor [2] the other two control flows monitoring also takes a small penalty from 1.77% to 4.94%. The performance penalty using an unpipelined PHOTON-1 with a latency of 132 cycles is substantial. And the penalty exploiting an unpipelined SPONGENT-1 (latency of 45 cycles) is the lowest, but still considerable.

Fig. 8. Average CPI of the secure processor with unpiplined and piplined hash engines

We also test the average CPI with pipelined lightweight hash engines. With pipelining, the difference of average CPI between different monitors based on the lightweight hash engines is negligible, the maximum average CPI increases no more than 6.36%, Fig. 8 also gives the average CPI of pipelined SPONGENT-1, which increases 4.83%. Moreover, with the use of pipelining, there is no obvious performance difference between the traditional and lightweight hash functions in average CPI. The proposed monitor and Arora's monitor performs similarly at instruction execution cycle, and better than Patel's (6.6%) [3] and Ragel's monitors (11.9%) [4].

4.2 Security analysis

The security of hash functions largely depends on the length of the hash value. The theoretical security boundaries are concluded by Eq. (1) and listed in Table I. We make random collisions of the hash value in the monitor model with 10,000 data. Fig. 9 shows the hash collisions of different lengths of hash value with different hash functions.

Fig. 9. Hash collisions of different monitors

MD5 and PHOTON performs a few better than SPONGENT and LHash when the length of hash value is 16-bit, the reason may be that the more encryption rounds the better the hash function performs in diffusion and confusion, and the output distribution of hash function is tending to a more uniform distribution. With the length of hash value increasing, the probability of hash collision is getting lower and the difference of different hash functions collisions is insignificant. When the length of hash value is 20-bit, the probability of hash collision is no more than 0.19%. Although the real security of collision resistance is not as high as the theoretical value ($2^{n/2}$), the probability of hash collision drops to a fairly low level ($<0.01\%$) when the length of hash value increases to 32 bits. The hash value length of our monitoring model is 16~32-bit, longer than the length of Arora's (16~20-bit) model [2]. Moreover, MD5 is broken by Xiaoyun Wang at Crypto 2004 (the 24th Annual International Cryptology Conference), not suitable for integrity detection any more. However, the lightweight hash functions SPONGENT, PHOTON and LHash are not found obvious loopholes to be broken so far. Therefore, the proposed monitoring model performs better than Arora's monitor in collision resistance. Besides, as exploiting 16-bit hash value, our monitoring model only needs 48 bits to record the monitoring information, is half of the monitoring model in [6], and effectively prevents the instruction tampering.

In Fig. 10 the code segment clearly includes three basic blocks: [0x1ddfc~0x1de58], [0x1de5c~0x1de88] and [0x1de7c~0x1de88], as framed by different colors. The start address of [0x1de7c~0x1de88] is from the jump target address of [0x1de54]. Malicious attacks are imitated by deliberately tampering designated data or instruction. Fig. 9 describes various attacks, such as tampering instruction, jump entry address and stop address. Table II gives a summary of all exception messages in the case of violation. As property information is fine-grained, and consists of instructions and start-stop address of basic block. Most software attack reflects these information, thus the proposed hardware-assisted secure monitor can theoretically resist almost any software attack form of tampering instructions or control

flows. By injecting specific errors, the hardware-assisted secure monitor is in a position to find out all differences from the static integrity monitoring model to avoid potential attacks. Through 10,000 injection attacks for each monitor with different lightweight hash function, the accuracy of integrity detection is more than 97.76% with 16-bit hash value, and 99.99% with 32-bit hash value.

Fig. 10. Examples of authentic monitoring process and malicious code attacks with secure monitor based on SPONGENT-88/80/8

Table II. Summary of exception message

No.	Exception situation	No.	Exception situation
0x01	Count overflow at the end	0x15	Count overflow before the end
0x02	Error of hash checksum	0x14	Timeout of basic block execution
0x03	Error of count checksum	0x21	PC value out of range
0x04	Error of basic block address (early)	0x22	Error of PC value
0x05	Error of basic block address (delay)	0x31	Storage search behind CPU
0x13	Error of skipping jump instruction	0x32	Monitor behind CPU

5 Conclusion

The hardware-assisted run-time monitor based on the lightweight hash function is proposed in this paper. The fine-grained property information just includes the basic block and the start-stop address of permissible program for simplifying monitoring information. Moreover, we improve some components of the lightweight hash functions in hardware implementation, and exploit the function as the integrity verification algorithm to further reduce resource cost of the whole monitor in the embedded secure processor. Then we build an OR1200-FPGA platform to confirm validity of the proposed monitor. Take five standard benchmarks for experimental objects, the results depict that the monitors account for no more than 8.37% area overheads in the entire processor. The average CPI of our secure processor with pipelining lightweight hash functions increases less than 6.36%. Through injection errors, the monitor is proved to be resistant to a wide range of software attacks, and the accuracy of detection is over 97.76%.

Acknowledgments

This research is supported by the National Science Foundation of China (Grant No. 60973106), the Key Project of National Science Foundation of China (Grant No. 61232009), Astronautic Support Fund (Grant No. 377054).