Instruction-level Real-time Secure Processor Using an Error Correction Code

Seok Min YOON, Seung Wook LEE, Jong Kang PARK, Jong Tae KIM
School of Electronics and Electrical Eng., Sungkyunkwan Univ., 300 Cheoncheon-dong Jangan-gu, Suwon, Gyeonggi-do 440-746, South Korea
jtkim@skku.edu

***Abstract*—In this paper, we present a processor that detects security-attacks at the instruction level by checking the integrity of instructions in real time. To confirm the integrity of the instructions, we generate a parity chain of instructions and check them at run time. The parity chain is generated using an error correction code used in a digital communication system, and the integrity checker has the same function as the error-detector module of the error correction code. This architecture can readily be applied to a general processor, because the checker is located between the processor core and the instruction memory. Compared with other cipher modules with the same key space, our instruction integrity checker achieves a faster check speed and occupies a smaller area.**

***Index Terms*—secure processor, security, instruction, correlation, chain.**

I. INTRODUCTION

The security and integrity of embedded systems have become more crucial to most of mobile-users today [1,2]. Besides several methods of attacking the hardware itself, most attacks are performed using instructions injected by the attacker [3-8]. Through such an attack, the attacker can obtain information or control authority of the system. In [9], we proposed an instruction-level security technique by generating and checking the parity of instructions, allowing us to prevent the execution of unwanted programs or instructions on a system. It can create a high level of security against attacks at the instruction-level, because a malicious attacker trying to manipulate the instructions would need to change not only the parity of the instructions, but also the entire parity-chain. It is also advantageous for secured software development processes that require more complex steps than the in existing studies [10-13]. In this paper, we present a secure processor that detects instruction-level tampering using an error correction code and present the implementation results. The manipulation or insertion of unauthorized instructions might make the system failures or unwanted actions. Because it can be regarded as data errors in communication systems, we applied the Reed-Solomon (RS) code, which is widely used in digital communications to generate the parity of instructions for integrity checks.

II. SECURED PROCESSOR ARCHITECTURE

The proposed security processor architecture is shown in Fig. 1. It has additional parity memory and an instruction integrity checker between the processor core and instruction memory. The instruction integrity checker reads the instruction and parity from both memories and checks the integrity. If the integrity of the instructions is broken, the checker notifies the user in a separate process. Because instruction memory is located outside the processor core in a general processor, our method can be easily applied to general processor architecture.

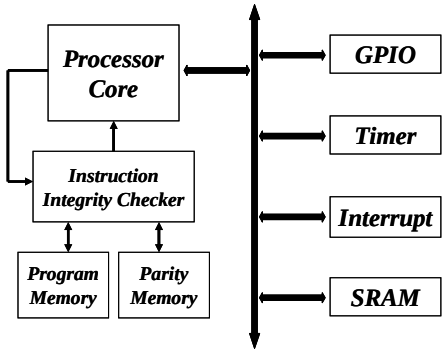

Figure 1. Architecture of processor with integrity checker

A. Instruction integrity checker

We exploit the error detection function of an error correction code as an instruction integrity checker. The distortion or insertion of instructions through a security attack can be considered a received data error in a digital communications system. We used the one of systematic codes, RS code in which the check can be done within one clock cycle instead of using a cipher module. The integrity checker inspecting the instruction errors, conducts the syndrome calculation in RS code. Even if the RS code is the one of error-correction codes, its syndrome calculation simply indicates whether the codes involve the errors or not [14,15]. That is sufficient for malicious code detection where any correction features are not necessary in the proposed secured processor. The RS code usually calculates the syndrome using a systolic array in a pipeline, because it receives data sequentially. However, the processor performs instructions on every single clock, and the check must be done in every single clock. We thus made a constant multiplier of the Galois field (GF) inside the integrity check

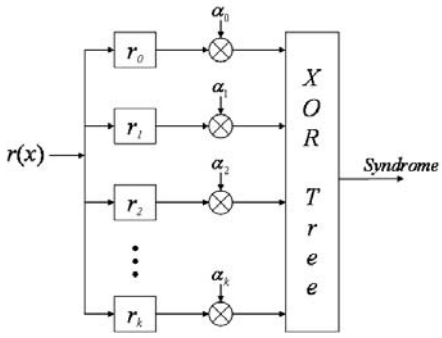

Figure 2. Architecture of the hardwired instruction integrity checker

This work was supported by IDEC (IC Design Education Center).
Digital Object Identifier 10.4316/AECE.2015.03002

in parallel, so that the calculation by the sophisticated logic circuit is done in a single clock [14] as shown in Fig. 2.

B. Parity chain generation process

The parity chain for checking instruction integrity consists of a parity chain of instructions and a parity chain of parities. Fig. 3 shows the process of parity chain generation. Parity1 (p_1) and parity2 (p_2) are parity of the instruction and the parity of the instruction parity, respectively.

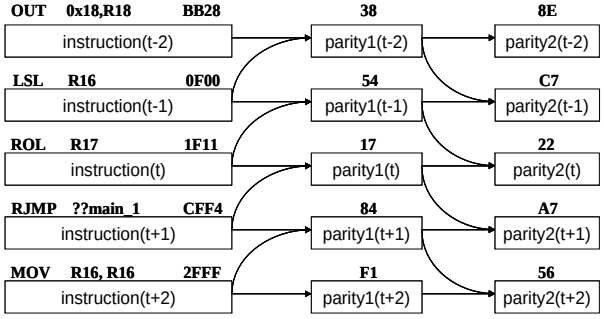

Figure 3. Instruction-level correlation structure

The parities in Fig. 3 are computed as follows [15]:

$$p(x) = x^{n-k} m(x) \bmod g(x) \tag{1}$$

where n denotes the length of a RS code and k is the length of parity words included in the code, where $n \leq k$. In eq. (1), $m(x)$ is the source data, regarded as a polynomial, and $g(x)$ is the generator polynomial of the RS code. The parity-generation in Fig. 3 can be expressed as follows:

$$p_1(t) = (x^{n-k} \cdot inst(t) + x^{n-k-1} \cdot inst(t+1)) \bmod g(x) \tag{2}$$

$$p_2(t) = (x^{n-k} \cdot p_1(t) + x^{n-k-1} \cdot p_1(t-1)) \bmod g(x) \tag{3}$$

The parity chain structure based on RS codes provides the security against the unknown attackers as the other typical cipher algorithms do. We can think of the specification of the RS code as a secret key, because the result of the parity calculation depends on the specification of the RS code. According to (1), parity is calculated using the Galois field, the generation polynomial, and input data. The generation polynomial is determined by the Galois field. When we calculate parity using GF(2^8), the number of candidate primitive polynomials is 16 and the number of generation polynomials is 255. The most important specification is the input data. Calculation of the RS code is applied to each polynomial coefficient. If we allow an input sequence with length 32 symbols using a set of 8 symbols, we can construct $8^{32}=2^{96}$ input data sequences. When we calculate parity using the above specifications, the number of RS code candidate specifications is 2^{108}. It is thus equivalent to having a 108-bit secret key. The key point is not only the computational complexity of the parity generation method but also the interdependency of parities. As shown in Fig. 3 and eq. (3, 4), the calculations of parity1 and parity2 at the current time t, are dependent on the previous, the current and the next instructions, create a chain structure. If an attacker tries to inject an instruction or manipulate the system, a parity error occurs, because previous parity was not computed with the data injected by the attacker. Because the parity codes have a correlation with each other, an attacker who tries to manipulate part of the parity results must change all of them.

III. SAFETY ANALYSIS OF INSTRUCTION-LEVEL CORRELATION TECHNIQUE

Conventional secure processors enhance security of a program by embedding cipher modules inside the processors. The safety of the instruction-level correlation technique depends not only on the complexity, but also on the management policy of the target system and the operation mode. Thus, the evaluation method for the proposed technique in terms of safety should be varied against that of the cipher algorithm itself. In this section, we discuss the safety of the proposed secure instruction technique where the one of RS codes is used as an integrity check for the instruction and its parity codes. The safety of security system relies on the application policy. Even if a certain cryptographic technique is logically, mathematically and stochastically perfect, it is hard to guarantee the entire system security when breaking down the management policies. The policies of the proposed instruction-level correlation technique can be defined as follows:

- Policy (1) : The security model with instruction-level correlation technique follows the model of [20].
- Policy (2) : Only an authorized user can access the protected part of system.
- Policy (3) : Only an authorized user can execute the secure process of the system.

In this section, we will discuss the safety of the proposed technique in conjunction with these policies above.

A. Safety of the instruction-level correlation technique

We apply RS code [15,16] to our system, and analyze its safety. Both the encoding and decoding processes of RS code are performed in the Galois field. The encoding process begins with a defining generator polynomial $g(x)$. Generally, a generator polynomial $g(x)$ is defined as eq. (4).

$$g(x) = \prod_{i=0 or i=1}^{i=2t-1 or i=2t} (x - \alpha^i) \tag{4}$$

where $t=(n-k)/2$. $g(x)$ creates parity information in the encoding process, and detects errors in the decoding process. After a generation polynomial is determined, additional parity is calculated as follows:

$$p(x) = x^{d-1} m(x) \bmod g(x) \tag{5}$$

where $d=n-k+1$ and $m(x)$ is the encoding data. Parity $p(x)$ is derived by the division of $m(x)$ by $g(x)$.

B. Complexity for inferring the code specification

In this sub-section, we analyze the mathematical complexity of inferring the specification of the RS code when it is used for extracting the method of correlation in our secured processor. The specification of the RS code includes the sequence of data in the encoding process. Fig. 4 shows an example of instruction-level correlation structure.

A malicious attacker can easily acquire secured data by tapping the external bus line of memory. Initially, attackers

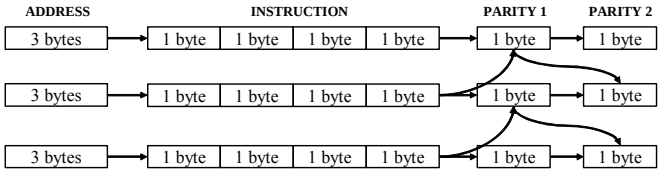

Figure 4. Instruction-level correlation structure for complexity analysis

try to infer the specification of the RS code from acquired data. Prior to analyzing the probability of inference, we setup two assumptions.

- Assumption (1) : A symbol of the RS code has 1 byte.
- Assumption (2) : The maximum data size for calculation of parity1, $p_1(t)$ is 256 bytes.

Assumption (1) concerns the basic size of the RS code. Calculation of the RS code is achieved by Galois's field of GF(2^8). Even if the symbol size of RS code can be more than 1 byte, this is one of the possible RS code candidates for the proposed correlation technique. Assumption (2) is made to limit the size of the RS code. Theoretically, there is no upper limits for data sizes, but this determines the complexity of the implementation stage. We consider the possibility of successful attacks in several conditions. A malicious attacker may have the following specifications.

- Specification (1) : The generator polynomial, $g(x)$
- Specification (2) : The number of data used in the encoding process
- Specification (3) : The order of data used in the encoding process

It is very hard to infer the specifications when the attacker only knows the given RS code used in the encoding process. The probability of inference is shown below.

- Inference probability for specification (1) : 1/65280

With single error correction capability, there are 65,280 of possible different generator polynomials in GF(2^8) using eq. (4). This is identical to the combination probability of $g(x)$ that has two roots.

- Inference probability of specification (2) : 1/252

According to the assumption that maximum data size is 256 bytes, there exist 252 different data sizes, because the data size can range from 3 (picking up an arbitrary 1 byte at each time $t(i\text{-}1)$, t, $t(i\text{+}1)$) to 255.

- Inference probability of specification (3) : $1/m^k$

As shown in Fig. 5, at time $t(i\text{-}1)$, t, $t(i\text{+}1)$, each data has m possible unit data and the size is k, and then the maximum m^k of the construction orders exist. Thus, if the number of unit data is increased, the possible number of the construction sequences increases exponentially.

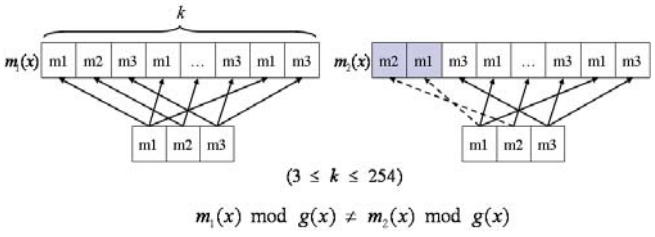

Figure 5. Construction of RS code

- Inference probability of all specification: 1 / 5.0x10^{15}, where m=3 and k=20 in specification (3)

Consequently, it is very hard to infer the exact code specification when the attacker only knows the exposed data constructed by the RS code. This is the same as an attacker knowing the cipher text and its construction algorithm in a conventional cryptography system.

C. Security analysis under the code exposure

We consider the cases when the code specification is exposed to the attackers in this sub-section. This situation is similar to that when the secret key has been leaked from a cryptography system. The attacker who knows the specification of the RS code can alter the parity of the whole program and may modify some parts of program. To address this problem, we redefine eq. (5) as eq. (6).

$$p = m \bmod g \tag{6}$$

The attacker who knows the RS code specification also knows as well as the exact m and g. Then, p is easily calculated. In contrast, if the attacker only knows p and g, it is very hard to calculate m because the modulo function is a one-way operation. The attacker may search for candidates for m, but finding the exact m is very hard. If p, m, and g exist as a specific number in the same number system and the range of these numbers are also known, then the candidates of m can be easily constructed by eq. (7).

$$m = ga + p, \quad (\min < m < \max) \tag{7}$$

However, in RS code, p, m and g are polynomials. These consist of the elements of combination, which are coefficients of polynomials in a finite field. They cannot be calculated by simple combination as in eq. (7). Instead, the attacker has to estimate the candidates of m by the polynomial, as shown in eq. (8).

$$m(x) = g(x)a(x) + p(x) \tag{8}$$

Here, the RS code can be represented as follows:

$$C(x) = g(x) \cdot Q(x) = d(x) \cdot X^{x-k} + p(x) \tag{9}$$

Every root of g(x) is also a root of C(x). Every coefficient of the polynomial is defined in Galois's field. We assume that RS(7,4) code is applied to our system. If the attacker knows $g(x)$ and $C(x)$, they can easily obtain $p(x)$ by dividing $C(x)$ by $g(x)$. Then, $d(x)$ and $p(x)$ are computed in this case. Conversely, if the attacker only knows $g(x)$ and $p(x)$, he needs to estimate $C(x)$ by eq. (9). Especially, the degree of n-1 has to be same with the highest degree of multiplication of $d(x)$ and $g(x)$. Thus, the highest degree of $C(x)$ should be x^6 and $d(x)$ should be x^3 when n=7. Now, we will see whether the attacker can figure out $d(x)$.

$$\begin{aligned} &ax^6 + bx^5 + cx^4 + dx^3 + ex^2 + fx + g = \\ &(hx^3 + ix^2 + jx + k)(x^3 + x + 1) + 2x + 2 \end{aligned} \tag{10}$$

The unknown variables can be reduced to this.

$$e + b + d + f + 2 = 0 \tag{11}$$

As a result, because there are more unknown variables than the number of equations, it is very hard to figure out correct unknown variables, even if the quantitative number of specification of RS(7,4) is very small. Whereas the number in the code specification is increased, the number of unknown variables should be increased. Nevertheless, another counterplan improving the security for this situation is to use magic number M in our system. The magic number M can be regarded as a secret-key in a cryptography system. In this approach, the first secret-key is the specification of RS code, and then we can consider magic number M as a second secret-key. In case of exposure of the important parameters to the attackers, a magic number can be a countermeasure as follows:

$$m' = f(m, M), \; p' = m' \bmod g \tag{12}$$

where m' denotes the modified original data by the function f and M. The attacker does not know the magic number M. In the secured program development process, the developers generate parity with eq. (5), after they perform function f with magic number M. The possible information that the

attacker knows is g, m and newly generated p'. This can have the same effect as if m is unknown. Next, assume that the attacker tries to infer magic number M when the attacker knows m. When f converts $m'=m^M$, eq. (12) can be re-defined as in eq. (13).

$$p' = m^M \bmod g \tag{13}$$

The attacker has to infer the magic number M, thus eq. (13) can be regarded as the *discrete logarithm problem* as shown in conventional cryptography. Consequently, in order to improve the security of the proposed technique, the use of magic number M can be one possible method.

IV. Experiment

The instruction integrity checker was implemented in VHDL and inserted in an Atmega103 download from OpenCore. As shown in Fig. 1, the integrity checker was located between the processor core and the parity memory. It reads instruction and parity values directly from memory and checks the integrity instruction. In this experiment, the instruction parity was stored in extra parity memory, because the AVR processor had the only instruction memory. Then, p_1 was generated using RS(10,8), and p_2 using an RS(6,4) code specification in the Galois field GF(2^4). Because the Atmega103 processor has a 16-bit instruction length, p_1 and p_2 had lengths of 8 bits each. The designed security processor was synthesized in XILINX ISE and programmed in xc2v6000 FPGA. Table I shows the result of the synthesis of the secured processor. To compare our instruction integrity checker with other cipher modules, we lists the synthesis results of both our integrity checker and AES, one of cipher algorithms in Table II. Our integrity checker requires 170 slices and 1 clock checking period; by contrast, AES requires 2784 to 5810 slices and a 10-to-50-clock delay for decryption. Thus, our instruction integrity checker occupies a small area (<12%) and has a faster check speed than other cipher modules.

TABLE I. IMPLEMENTATION RESULTS

	Atmega103	Secured Atmega103
Number of Slices	1354	1458
Number of Slice Flip Flops	838	999
Number of 4 input LUTs	2598	2792
128×16 ROM	1	2
Maximum Clock Speed	46.192 MHz	49.387 MHz

TABLE II. COMPARISON WITH OTHER CIPHER MODULES

	Design	Device	Slices	Cycles
Standaert [17]	AES Encryption	XCV3200E	2784	21
Saggese [18]	AES Encryption	XVE2000	5810	50
Wnag [19]	AES Encryption	XCV1000E	5150	10
Our Checker	Integrity checker	XC2V6000	170	1

V. Conclusion

In this paper, we present a secure processor architecture that embeds a single-cycle and low hardware overhead (<12%) integrity checker for the parity chain. To check the correctness of an instruction set, we generate a parity chain using RS Code and store it in separate parity memory and the checking of instruction integrity proceeds. According to experimental results, the proposed architecture can easily be implemented in a general processor and it checks the integrity of instructions in real-time. Compared with other cipher modules with the same key space, our checker exhibits faster check speed and requires a smaller area.

References

[1] M.L. Pollar, F. Martinelli and D. Sgandurra, "A Survey on Security for Mobile Devices," IEEE Comm. Surveys and Tutorials, Vol.15, No.1, pp.446-471, 2013. doi:10.1109/SURV. 2012.013012.00028.

[2] K. Nikita, "Security and Privacy in Biomedical Telemetry: Mobile Health Platform for Secure Information Exchange," Wiley-IEEE Press eBook Chapters, 2014. doi:10.1002/9781 118893715.ch13.

[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in Embedded Systems: Design Challenges," ACM Trans. on Embedded Computing Systems, Vol. 3, pp. 461-491, 2004. doi:10.1145/1015047.1015049.

[4] A. Murat Fiskiran, Ruby B. Lee "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution" ICCD, 2004. doi:10.1109/ICCD.2004.1347961.

[5] T. Maude and D. Maude, "Hardware Protection Against Software Piracy", Communications of the ACM, vol. 27, no. 9, pp.950-959, 1984. doi:10.1145/358234.358271.

[6] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software", Proc. of ASPLOS, pp. 168-177, 2000. doi:10.1145/378993.379237.

[7] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Sheperding," Proc. of 11th USENIX Security Symp., pp.191-206, 2002.

[8] D.L. Detlefs, K. Leino, G. Nelson, and J. Saxe, "Extended static checking," Tech. rep., Systems Research Center, Compaq Inc., pp.1-44, 1998.

[9] S.W. Lee and J.T. Kim, "Instruction Level Tampering Detection Technique using Error Detection Code for Embedded Systems," Proc. of ICCMSE, Vol. 1, pp.1-4. 2005.

[10] W. Arbaugh, D. Farber, and J. Smith. "A secure and reliable bootstrap architecture" Proc. of IEEE Symp. on Security, pp.65–71, 1997. doi:10.1109/SECPRI.1997.601317.

[11] G.E. Suh, D. Clarke, B. Gassend, M.v. Dijk, S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", Proc. of IEEE/ACM Int'l. Sym. on MICRO, pp. 339-350, 2003. doi:10.1109/MICRO.2003.1253207.

[12] J.P. McGregor, D.K. Karig, Z.Shi, and Ruby B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks", Proc. of IEEE Int'l. Conf. ITRE, pp. 243-250, 2003. doi:10.1109/ITRE.2003.1270612.

[13] B. Gassend, G.E. Suh, D. Clarke, M.v. Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", Proc. of Int'l Symp. on HPCA, pp.295-306, 2003. doi:10.1109/HPCA.2003.1183547.

[14] S.T.J. Fenn, M. Benaissa, and D. Taylor, "Bit-serial Berlekamp-like multipliers for GF(2^m)," Electronics Letters, Vol. 31, 1995, pp. 1893-1894, doi:10.1049/el:19951303.

[15] I.S. Reed and X. Chen, "Error-Control Coding for Data Network," Kluwer Academic Publishers, 1999.

[16] S.W. Lee, J.T. Kim and J-S. Cha, "Implementation of Adaptive Reed-Solomon Decoder for Context-Aware Mobile Computing Device," LNCS, Vol. 3681, 2005. doi:10.1007/11552413_158.

[17] F-X. Standaert, G. Rouvroy, J-J. Quisquater, and J-D. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs," LNCS, vol.2779, pp. 334-350, 2003. doi:10.1007/978-3-540-45238-6_27.

[18] G..P. Saggese, A. Mazzeo, N. Mazzocca, and A.G.M. Strollo, "An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm," LNCS, vol.2778, pp.292-302, 2003. doi:10.1007/978-3-540-45234-8_29.

[19] S-S. Wang, and W-S. Ni, "An Efficient FPGA Implementation of Advanced Encryption Standard Algorithm," Proc. of ISCAS 2004, Vol. 2, pp. 597-600, 2004. doi:10.1109/rivf. 2012.6169845.

[20] W. Shi, H-H. S. Lee, C. Lu, and M. Ghosh. "Towards the Issues in Architectural Support for Protection of Software Execution." Proc. of ASPLOS, pp.1-10, 2004. doi:10.1145/1055626. 1055629.