Safe Microcontroller with Silent Hooking-Unit of Runtime Bus Transition for Code Memory Integrity Verification

Hyeonjae Shin1,2, Jeonghun Cho1 and Daejin Park1*
Postal: 41566, Daehak-ro 80, Buk-gu, Daegu City,
1School of Electronics Engineering, Kyungpook National University,
2LG Electronics, Republic of Korea.
*Corresponding Authors
ORCID: 0000-0002-5560-873X, Scopus Author ID: 55463943600

Abstract
The microcontroller is controlled by software code execution in embedded flash memory, so small errors or bit inversion during instruction-fetch operations causes unexpected harmful malfunction of the entire system. Explicitly, user-defined code insertion to verify the instructions requires more implementation efforts in writing check codes and additional areas in code memory. In addition, during the software code execution for memory integrity verification, the main service routines will be delayed or out of service. This paper introduces our experience of integrating the hardware-assisted memory integrity verification unit into the on-chip code memory bus architecture. To reduce the time-delay penalty during the integrity verification, we adopted the hooking method to steal the bus idle time, and then we could collect small fractions of code memory data to build an expected cyclic redundancy check (CRC)-based fingerprint of the entire code memory data. In this paper, especially for flash instruction memory, we propose silent execution hardware calculating the CRC of the ROM data during the CPU idle time without any CPU wait cost. Parallel CRC hardware and the dedicated finite state machine (FSM) are executed faster at the clock level compared to software CRC calculation that requires an explicit CPU wait state. The 64KB ROM integrity verification in the experimental 8051 MCU requires about 24ms of execution time at an 8Mhz clock speed without any CPU wait state when running silently in background mode, and it requires an additional 1650 gates for the proposed hardware data path.

INTRODUCTION AND MOTIVATION
The embedded systems are normally implemented using commercial off-the-shelf microcontrollers with on-chip code memory. The system functionalities are controlled by the embedded software in the on-chip flash memory. The internal core of the microcontroller (MCU) accesses the code memory to fetch instructions and then interprets the software-directed functions. If small bit error of the embedded binary code in the code memory occurs, the system does not guarantee compliance in terms of special safety requirements.

The system integrator and software implementer always take care of the abnormal instruction executions due to the unexpected fetch operations from the code memory. Fig. 1(a) shows the programmer’s effort in considering safe code execution by inserting a more careful check code. This approach requires explicit code insertion to verify the instruction fetch and to compare the expected patterns, causing more area in the code memory. In addition, execution of manual check code for the code integrity verification results in the inevitable halt of CPU’s main software.

Fig. 1(b) is an illustrated concept of our approach to resolve these issues raised by the conventional method in Fig. 1(b). We integrate the hardware-assisted verification unit [1][2] into the on-chip code memory bus architecture. This approach reduces additional effort in preparing the code integrity check codes and does not require inevitable intervention of the CPU’s main software code executions. The hardware-based integrity verification approach is relatively faster than the result of CPU-based method and it also gives us more transparent operations because system engineers do not care about the code memory integrity issues.
IEC-60730 [3] specifies all the requirements for the safe operation of digital MCU. We summarized the IEC-60730 compliance specification for safe operation requirements in Fig. 2. In particular, we summarized as much because fetch failure of the instruction memory can cause an abnormal operation or the system to remain stuck in a dangerous state, thus overheating the system. IEC-60730 defines mainly memory-related tests [4]. Due to test cost issues, most MCU vendors only support a memory test in the software mode to reduce the silicon area by halting the main user code execution. The safe MCU, which is implemented for this paper, has the features equipped with the hardware and software approaches in the red box. Fig. 3 describes the hardware-assisted approaches to protect these system malfunctions. This paper focuses on performing the integrity verification of code memory in Fig. 3(c).

The remainder of the paper is organized as follows: The related works are discussed in Section 2, the proposed architecture and its methods are described in Section 3, the implementation and evaluation results are shown in Section 4, and the conclusions are presented in Section 5.

2 Related Works

Some MCUs [5][6] provide a software stack supported by the hardware module for easy programming. The ARM Cortex-M MCU offers a fault monitoring co-processor to detect any CPU malfunction [7]. A fault monitoring MCU is also proposed for automotive applications [8]. Some MCUs supply an error check code (ECC)-based protection method for flash memory [9]. One study shows a method to reduce ROM access current based on the binary bit-inversion [10][11] for ROM encryption-based protection.

The runtime memory integrity validation during the CPU’s main software execution requires additional intervention such as CPU halt. In terms of waste-time, hardware resources for the special ROM section, including the ROM scanning subroutine and automatic FSM-based memory scanner, are required in the on-chip. The ROM integrity check technique, which is implemented according to the minimum time and space requirements, is an important issue in designing the most cost-effective MCU for the safe operation of applications.

Our previous work introduced the background mode CRC calculation-based code memory integrity verification [12]. We also improved this work by adopting the binary bit-inversion-based scrambling technique for the given instructions [13], which could then be integrated into the flash memory read-path bus architecture [14]. In this paper, we focus on low-cost code ROM protection with a CRC hardware acceleration block supported with the proposed silent calculation technique based on the on-chip resource reallocation concept to reduce the CPU delay penalty.

![Figure 2: IEC-60730 compliance specification for safe operation requirements](image)

<table>
<thead>
<tr>
<th>Protection Criteria</th>
<th>Target</th>
<th>What to test</th>
<th>This paper's hardware/software approach</th>
<th>Exception handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>System SFR protection</td>
<td>Registers, PC</td>
<td>Stuck, access violation</td>
<td>Hardware / software lock protection support</td>
<td>Reset, interrupt</td>
</tr>
<tr>
<td>RAM protection</td>
<td>RAM data, RAM address, Write lock</td>
<td>Access violation</td>
<td>Hardware checksum, parity protection (background mode)</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Flash ROM protection</td>
<td>ROM data ROM address</td>
<td>Access violation, ROM data integrity</td>
<td>Hardware CRC protection (background mode special block)</td>
<td>Reset, interrupt, backup/restore mode</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>Accuracy</td>
<td>Timing, stuck</td>
<td>Clock switch by backup clock</td>
<td>Clock switch or WDT reset</td>
</tr>
<tr>
<td>Communication</td>
<td>I/O</td>
<td>Corruption, stuck</td>
<td>Line watch-dog timer to resolve direction stuck</td>
<td>Peripheral reset, interrupt</td>
</tr>
</tbody>
</table>

![Figure 3: Various protection method](image)
**PROPOSED ARCHITECTURE**

Fig. 4 describes the architecture of the safe MCU with the proposed silent hooking-unit of runtime bus transition for code memory integrity verification. The MCU consists of four blocks: CPU, data memory in RAM, code memory in flash, and code verification unit. The CPU accesses RAM to fetch or write data value, and code memory to fetch instructions through the instruction bus. We extract the instruction bus and connect into our proposed verification unit, which is described in item (1) of Fig. 4. The finite state machine (FSM) of the verification unit controls the scanning functions by moving down the access location to fetch the instruction data during the CPU idle time. Without any support of software execution, the on-the-fly fingerprint value will be extracted and calculated using the CRC-based method, which is denoted in item (2) of Fig. 4. If the calculated fingerprint value for the given code memory data is matched with the expected value in item (3) of Fig. 4, we can estimate that the currently executed software code is safe, so that we can continue executing the instructions.

The proposed architecture including these micro-operations is designed as an easily pluggable and independent unit with the main CPU core, for easy configuration without requiring any interventions of original CPU functions. The conventional fingerprint calculation such as CRC requires explicit CPU intervention because software code execution still accesses the code memory, as shown in Fig. 5(a).

The periodical verification of the code memory results in the delay of original code execution, causing different execution results as a side effect. Instead of the conventional method based on CPU halt, the proposed fingerprint calculation unit, as shown in Fig. 5(b), steals the idle time of CPU operations, utilizing the fact that CPU does not always occupy the code memory bus. During the small fraction of the time slot, we access the code memory to collect the fingerprint value and build the entire calculated CRC value to determine whether the currently accessed code memory region is broken.

The hardware-based verification unit with independently pluggable insertion to verify the code memory will be efficiently applied during the CPU’s main software code execution. Fig. 6 describes the efficient execution of the proposed verification task. Our approach of collecting the small fraction of code memory enables easy pluggable verification of the code memory during the idle time slot to minimize the overhead in terms of execution time.
As a result, the proposed approach in the background mode reduces the total execution time including the original software code execution and additional verification by utilizing the wasting time due to the CPU idle slot. The hardware-accelerated verification method also reduces the active time in the system mode, so that the energy consumption will be slightly decreased.

The conventional interrupt service is asynchronously requested and requires explicit CPU wait time. Although there are idle slots in the CPU time, the CPU processing task must be split after the interrupt service call. This causes a delay in the CPU completion time, which results in a more active power mode with an analog power block of the power management unit (PMU). The total energy needed to manipulate the fixed task increases in accordance with the duration to the active mode.

The proposed architecture partially splits the request service by dividing the task into several segments in the CPU idle slot. In this paper, we focus on the ROM integrity verification with CRC calculation. This task can be split according to the fragments of the calculation task and efficiently allocated to the CPU idle time to utilize the CPU waste time. This method increases the utilization of the BUS access time and decreases the completion time so that the chip can quickly enter the sleep state to reduce the power consumption by turning off the analog power circuit.

To increase the efficiency of the proposed technique, more CPU idle slots are needed for efficient reallocation to minimize the delay of the on-demand interrupt requests. We adopt the hardware acceleration method to shorten the CPU completion time and create additional CPU idle slots. Although more CPU idle slots require an increase in the CPU speed, the entire power consumption will be recouped by decreasing the DC current in the analog circuits, such as PMU and the voltage down converter.
IMPLEMENTATION RESULT

Fig. 7 describes our implemented MCU. CPU always checks the code memory integrity using the proposed verification unit in the instruction bus level without any CPU halt during idle time. The programmer writes down the user-defined software and compiles it into the binary code to be downloaded into the on-chip code memory in flash memory.

Our approach requires additional padding procedures to prepare the fingerprint for the given binary code. The binary code will be scrambled with the specific function to determine whether the currently-accessed binary code is broken. The proposed approach needs slight modification of additional compilation phases and areas to hold the calculated fingerprint into the flash memory.

The proposed hardware, based on the finite state machine, the CPU idle detector, and parallel CCIT-16 CRC hardware is implemented on the Mentor 8051 synthesizable core with a total additional 1650 gates in a 0.18\mu m CMOS process. The hardware overhead is a reasonable size compared to the code memory area required for the CRC calculation of the software.

The Mentor 8051 CPU core does not always access the flash ROM memory. For the experimental core, one out of every...
three clocks is an idle slot to use the flash memory BUS interface to read the ROM data in the specified address.

Fig. 8 depicts the implementation details for the hardware CRC calculation block core. The cyclic operation for CRC calculation, which requires a faster shift operation, is implemented with the parallel CRC data path performed in one clock. The iterative CRC calculation per fetched data is performed and updated with the feedback from previous calculations of CRC data. Fig. 8(a) represents the CRC calculation cycle loop operation and Fig. 8(c) is a fraction of the implemented CRC pseudo code.

Fig. 8(c) shows the simulated result of the on-the-fly replacement of the current CPU address to access the flash memory silently during the CPU idle time without any CPU interruption that requires a delay in the task completion time.

During normal CPU execution at an 8Mhz clock speed, the proposed hardware finished the CRC calculation with the silent background-mode hardware in 20-30ms to verify all 64kB ROM data. The conventional CRC calculation method for the experimental MCU required about 125ms in the software mode and about 10ms using fully burst CRC calculation hardware support.

CONCLUSION

Code memory integrity verification hardware architecture for safe MCU is proposed using the dedicated hardware based on the CRC calculation in the background mode. The proposed approach reduces the CPU intervention such as the halt of original software code execution, by stealing the CPU idle time slot. The proposed verification unit collects the small fraction of code memory to build the calculated fingerprint so that we could transparently determine whether the currently accessed code is broken or not. The additional hardware cost is slightly reasonable compared to the advantages, which do not require explicit code insertion complexity and design efforts in writing test code to verify the code memory.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest regarding the publication of this paper.

ACKNOWLEDGMENT

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (2014R1A6A3A04059410 and 2016R1D1A1B03934343). Corresponding author is Daejin Park (boltanut@knu.ac.kr).

REFERENCES AND NOTES


