A Software Approach to Protecting Embedded System Memory from Single Event Upsets

Jiannan Zhai, Yangyang He, Fred S. Switzer, and Jason O. Hallstrom

Clemson Univeristy
{jzhai,yyhe,skyler,jasonoh}@clemson.edu

Abstract. Radiation from radioactive environments, such as those encountered during space flight, can cause damage to embedded systems. One of the most common examples is the *single event upset* (SEU), which occurs when a high-energy ionizing particle passes through an integrated circuit, changing the value of a single bit by releasing its charge. The SEU could cause damage and potentially fatal failures to spacecraft and satellites. In this paper, we present an approach that extends the AVR-GCC compiler to protect the system stack from SEUs through duplication, validation, and recovery. Three applications are used to verify our approach, and the time and space overhead characteristics are evaluated.

1 Introduction

One high-energy ionizing particle passing through an integrated circuit can release enough charge to change the state of a binary digit, causing a stored bit to change to its opposite value (i.e., a 0-bit can become a 1-bit, and vice-versa [18]). The results can range from system malfunction to system crash.

Modern approaches used to prevent and correct SEU errors often introduce additional hardware to the target system. In this paper, we present a *software-only* approach that detects and corrects SEUs in RAM. The paper focuses on the system stack, which is the most important and dynamic region in memory. The system stack is protected by injecting code into the target assembly generated by AVR-GCC. Our approach does not introduce additional hardware, and since it operates at the assembly level, it is language and application neutral.

2 Related Work

One of the primary hardware-level radiation hardening approaches is Silicon-on-Insulator (SoI) technology, used in microprocessor fabrication [1]. The design improves the circuit's tolerance to highly-charged particles, reducing the chance of SEU occurrence. Irom et al. [7] compare SEU error rates in SoI microprocessors to conventional microprocessors. SEU rates were observably lower in SoI microprocessors. Though SoI technology protects systems from SEUs, it prevents developers from using commercial off-the-shelf (COTS) devices, increasing system cost due to the high price of SoI circuits.

Redundancy is a widely used fault-tolerance technique, both via hardware and software. The Triple Modular Redundancy (TMR) [8] approach executes instructions on three unique systems. A voting module is used to compare the results and choose the common result. Due to the low probability that more than one SEU will occur simultaneously at the same geographic location in more than one device [16], TMR is a popular SEU protection technique and allows the use of COTS components. However, hardware-based TMR introduces significant hardware overhead and power consumption, which can present concerns for weight-limited and power-critical systems.

Time Redundancy [2] is a software-only redundancy technique which runs each instruction three times on a single processor. The results are stored, and a voting module is invoked to yield the (most) common result. Error Detection by Duplicated Instructions (EDDI) [11], a variation on Time Redundancy, duplicates each instruction during the compilation phase and assigns each different registers and memory space. As a result, EDDI is able to protect systems from not only data SEUs, but also instruction SEUs. Time Triple Modular Redundancy [2] is a combination of time redundancy and hardware-based TMR. Each instruction is executed by three unique systems, as in standard TMR, but the systems execute the instruction in different clock cycles in a time-redundant fashion. This allows more instructions to be executed in parallel.

A watchdog timer (WDT) [6] is a timer used to detect and recover from system crashes by repeatedly querying the protected system and resetting the system if no response is received. A software-based WDT is straightforward to implement and introduces little overhead. However, it suffers the risk that an SEU may cause the WDT itself to malfunction. Despite increased cost, hardware-based WDT provides a reliable solution. Note, however, that WDT is typically used with other techniques since it only detects severe system faults.

Shirvani et al. [13] examine a set of Error Detection and Correction (EDAC) methods used to detect and correct errors in memory, such as those caused by SEUs. The authors find that the reliability of software-based methods tends to decrease over time more rapidly than hardware-based methods. However, the loss rate is low enough that software-based methods are still more cost effective than hardware-based methods.

3 System Design

We focus on the protection of the runtime stack under the following assumptions: (i) Flash memory and registers are not affected by SEUs. (ii) Only one SEU will occur during a given function execution. It is rare for more than one bit to be upset simultaneously; this occurs in only 5 to 6 percent of bit flip errors [17].

Our approach is designed to align with the NASA coding standards for C applied in space projects [10]: First, dynamic memory allocation is not allowed, so the heap section in RAM is not used. However, for the sake of completeness, we consider the possibility of a non-empty heap in our approach. Second, the `goto` statement is not allowed. Finally, each function should have fewer than 60 lines of code, making the execution time of each function relatively short.

Our approach protects the system stack by introducing auxiliary assembly code. The new code is injected at both the beginning and end of each function or interrupt handler. When a function is called, the code injected at the beginning of the call calculates the CRC of the caller's stack frame and saves both the

CRC and the stack frame. Before the callee returns, the code injected at the end calculates the CRC of the caller's stack frame again, compares it with the saved CRC, and restores the caller's stack frame if the CRCs do not match.

3.1 Supporting Memory Sections

To store stack frame replicas, two new sections are created in SRAM just after the `.bss` section by modifying the linker script [14]. The new `md` section is used to store stack frame replicas, which are referred to as *Stack Frame Snapshots* (SFSs). The new `sp` section is used to store the address of the next available memory space in `md`, similar to the stack pointer. This address is referred to as the *Snapshot Top Pointer* (STP). To protect the STP from SEUs, the size of the `sp` section is set to 6 bytes, and 3 STP duplicates are stored in this section. Given that we assume only one SEU will occur during the execution of a given function, only one STP replica could be altered by a flipped bit. The altered STP is easily identified and corrected by comparing the values of the three STP replicas.

3.2 Injected Code Segments

The injected code segments are designed to use only registers, reducing their dependence on RAM. *CRC Calculate* is used to calculate the CRC of a memory region. In our implementation, CRC16-CCITT is used [5]. *CRC Save* is used to save the CRC to the stack. *CRC Compare* is used to compare two CRCs. The comparison result indicates whether an SEU is detected. *Frame Copy* is used to copy a stack frame to a given destination, and to save and restore stack frames. *Frame Size Save* is used to save the size of the stack frame for the current function. *STP Initialize* is used to initialize the STP so it points to the lowest address of the `md` section. *STP Update* is used to update the STP. First, it obtains the correct STP value by comparing the three STP replicas. The replicas are then updated to reflect the addition or removal of a stack frame.

3.3 Modified Function Execution Process

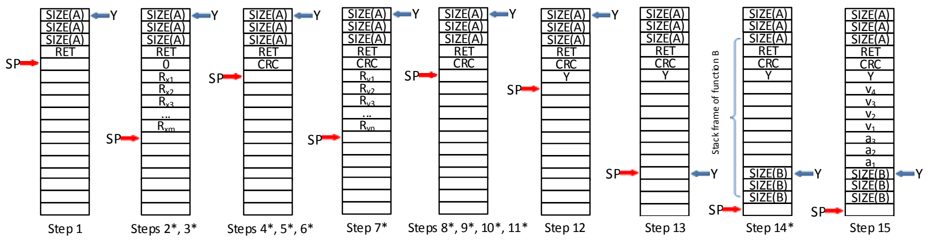

Fig. 1: Modified Function Invocation Process

Modified Function Invocation Process. The code segments injected at the beginning of each function are used to calculate a CRC over the caller's stack frame, and to save a duplicate of the caller's stack frame, as shown in Figure 1. Each rectangle represents two stack bytes. The "starred" steps denote stack changes caused by the injected code. `SP` denotes the stack pointer, and `Y` denotes the stack frame pointer.

When a function `B` is called by a function `A`, the return address is pushed onto the stack automatically by the function call instruction (step 1). To calculate the CRC of the caller's stack frame, multiple registers are used, so they must be saved before the CRC calculation process and restored when the process is finished. To prevent the calculated CRC from being overwritten when the registers are restored, two bytes are pushed onto the stack as a placeholder (step 2) for the CRC result before the registers used to calculate the CRC are saved (step 3). After the CRC of function `A`'s stack frame is calculated (step 4), the CRC result is saved to the placeholder location (step 5). The registers used to calculate the CRC are then restored (step 6).

Next, the stack frame of the caller, function `A`, has to be saved. The registers used to save the frame are pushed (step 7). Next, the correct STP is selected by comparing the values of the three STP copies (step 8). Using the correct STP, the specified memory is then copied and saved in the SFS (step 9). After the STP copies are updated (step 10), the CRC registers are restored (step 11).

After the stack frame pointer of function `B` is saved (step 12), and the stack frame is established (step 13), three copies of the callee's frame size are pushed onto the stack (step 14), which is a key operation in the injected code.

When a function is called, the callee does not have sufficient context regarding its caller, including the caller's stack frame address and size. It is impossible for a callee to calculate the CRC of it's caller's frame and to duplicate the frame without this information. To solve this problem, each function saves its frame size in the stack, which is used by its callee to perform the CRC calculation and frame copy. To ensure the correctness of the frame size, three copies are saved. Comparison is used to yield the correct value.

Modified Function Return Process. The code segments injected at the end of each function are used to verify the stack frame of the caller, and to restore the stack frame if an SEU is detected, as shown in Figure 2. When function `B` returns, it first pops its stack frame size (step 1). After the space used to store the arguments and local variables is released (step 2), the stack frame pointer is restored (step 3). The CRC of function `A`'s stack frame is then calculated and temporarily stored in two registers (steps 4-6). The values stored in these registers are saved before the function return process. Next, the calculated CRC is compared with the CRC saved in the stack (step 7). If the two CRCs do not match, the saved stack frame of `A` is restored, and the STP is updated to release the space used to store the stack frame of `A` (steps 8-12). Again, the stack frame size of function `A` saved in the stack is used to support the CRC comparison and stack frame restoration (if needed). If the two CRCs match, the STP is updated (steps 13-14). After verification of `A`'s stack frame is complete, the CRC is popped from the stack (step 15). Finally, function `B` returns, and the return address is popped automatically (step 16).

4 Evaluation

To evaluate our approach under varying stack conditions and SEU injection rates, three AVR applications are considered. The stack usage pattern of each application is shown in Figure 3.

4.1 Validation

We first validate our approach and consider the SEU protection efficacy it affords. In our analysis, we ignore both the `.data` and the `.bss` sections, as well as the heap section. Data stored in the `.data`, `.bss`, and `heap` sections can be protected using well-known techniques based on cloning and comparison. We focus our analysis on stack frame protection. We first assume that the currently

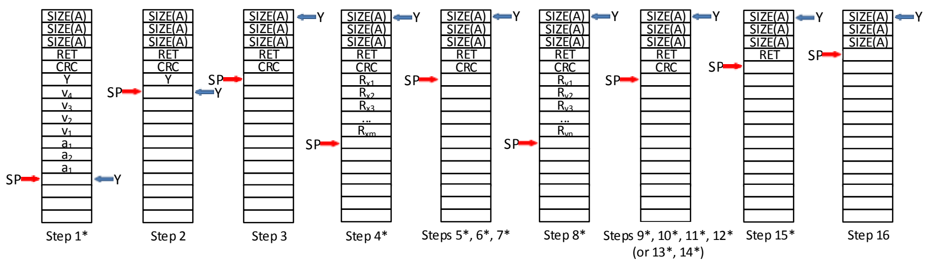

Fig. 2: Modified Function Return Process

executing function's frame, which includes the return address of the injected code segment, is not affected by SEUs. The stack frames of callers and callees are guaranteed to be correct, so the stack is guaranteed to be correct. To verify this claim, the AVR Simulator IDE [12] was used to manually inject SEUs, and to observe execution results. The results showed that each function is able to detect and fix SEUs introduced "beneath" the topmost stack frame.

However, if the stack frame of the current function is affected by an SEU, protection is not guaranteed. If the SEU changes key data, such as the return address or stack frame size, the current function will not execute as expected.

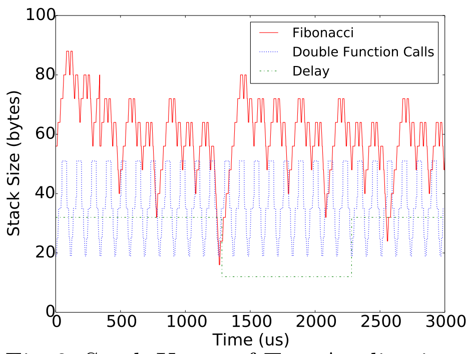

Fig. 3: Stack Usage of Test Applications

We assume that only one SEU will occur during a given function execution, and that the SEU is uniformly likely to affect all bits in RAM. The probability of successful SEU protection can be expressed as $p = 1 - c/(2s + e - c + 6)$. Where p is the probability of successful protection, s is the stack size, e is the size of the unused space in RAM, 6 is the size of the three STP copies, and c is the average size of a stack frame. Since the return address of the injected code segment is stored in the current stack frame, the two bytes for the return address are included in c. The total size of protected memory is $s + e + (s - c) + 6$, where $s - c$ is the size of the stack frame copies stored in the *md* section.

We extend our analysis to cases where more than one SEU may occur during a given function execution. Due to lack of space, we omit the derivation details. The (conservative) probability of successful protection can be expressed as:

$$\begin{aligned} p = {} & (1 - \frac{c}{2s+e-c+6})^n * (1 - \frac{6}{2s+e-2c+6})^n \\ & * (1 - \frac{6}{2s+e-2c})^n * \{(1 - \frac{2c}{2s+e-2c-6})^n \\ & + \mathrm{C}_2^1 * (1 - \frac{c}{2s+e-2c-6})^n * [1 - (1 - \frac{c}{2s+e-3c-6})^n]\} \end{aligned} \tag{1}$$

Where p is the probability of success, s is the size of the stack, e is the size of the unused space in RAM, 6 is the size of the three stack frame size copies or the three STP copies, c is the average size of the stack frame (including the return address of the injected code segment), and n is the number of SEUs that occur during a function's execution. In equation 1, the number of SEUs that occur, n, can be expressed as $n = y * l * f/m$, where y is the number of clock cycles used to execute each instruction, m is the frequency of the microprocessor, l is the average number of function instructions, and f is the SEU injection rate. Most AVR instructions require 2 clock cycles to execute, and the frequency of our ATmega644 is set to 10MHz.

We now consider the relationship between SEU protection probability and SEU occurrence rate. To demonstrate the relationship, we collect the corresponding parameters for the three test applications using AVR Simulator IDE. Figure 4 plots the change in SEU protection probability as a function of SEU injection rate. The x-axis represents the rate at which SEUs are injected, and the y-axis represents the corresponding SEU protection probability. Each vertical line marks where the number of SEUs begins to exceed 1 (for each application). When only one SEU occurs during a given function execution (left side of the vertical line), the SEU protection probability is constant (Delay: 99.48%, Double Function Calls: 99.71%, Fibonacci: 99.68%) because the only case the approach cannot handle is when the current frame is affected. When more than one SEU occurs during a given function execution (right side of the vertical line), the SEU protection probability increases. As the SEU occurrence rate increases, the SEU protection probability decreases, until it approaches 0. The lower the stack dynamism, the longer the function execution time, which increases the probability of SEU occurrence in the current stack frame.

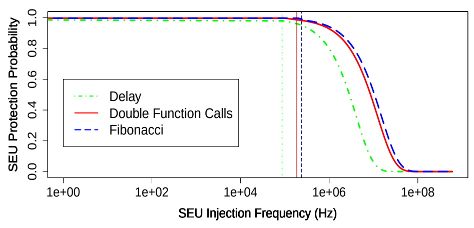

Fig. 4: SEU Protection Probability

4.2 Performance

Since the same code is injected for every function, the execution overhead is similar for all functions, varying only when an SEU is detected. The execution overhead depends on the size of the (recovered) stack frame. The *CRC Calculate* code segment and *STP Update* code segment execute twice for each function, and the *Frame Copy* code segment executes either once or twice, depending on whether an SEU is detected. Each of the other code segments executes once for each function execution. The minimum overhead introduced in terms of number of clock cycles is $62 * S + 304$, when an SEU is not detected. The worst case is $70 * S + 432$ clock cycles, when an SEU is detected.

We next evaluate space overhead using the three test applications. The ROM space data was collected using *avr-size*. The results are summarized in Figure 5. The y-axis represents ROM size, in bytes. Delay and Fibonacci involve two functions, and Double Function Calls involves four. From Figure 5, we can see that the ROM overhead for the Double Function Calls application is twice the Delay and Fibonacci applications. ROM overhead is related only to the number of functions in the program.

4.3 Physical Hardware

To validate our approach on physical hardware, we emulate the occurrence of SEUs by flipping random bits in the target SRAM area. To perform auditable test runs, we developed an AVR application which continuously generates an increasing integer sequence, which is then sent to the UART interface at a controllable speed. A Python program running on a desktop is used to receive the sequence and observe the impact of flipped bits by monitoring the continuity of the sequence. A timer interrupt is used to trigger the occurrence of SEUs. The interrupt service routine generates a random address within the range of the top of the stack and the end of RAM space, excluding the stack frame of the current interrupt, and then flips the bit at this location.

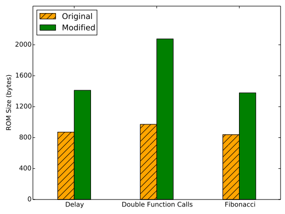

Fig. 5: ROM Overhead

We declare (observable) failure when one of the following two situations occurs: (i) The AVR application stops generating integers; or (ii) the integer sequence received by the Python program becomes discontinuous. We monitor the integer sequence and record the maximum count before failure. The experimental results are summarized in Figure 6. The x-axis represents SEU injection frequency, and the y-axis represents the maximum count received by the Python program. The figure shows that as SEU injection frequency increases, running time to failure decreases. This is explained as follows: As SEU injection frequency increases, the probability that an SEU occurs in a critical area increases. When the frequency is extremely high (e.g. approximately 10 MHz), the program can hardly send any values. However, the observed SEU occurrence rate in outer space is approximately $10^{-6} SEU/bit\text{-}Day$ [16]. Given that the total RAM size of an Atmega644 is 4K Bytes, the expected SEU occurrence rate for an Atmega644 is 0.0032 SEU/day, which is significantly lower than the lowest frequency (9765.625 SEU/second) that we used. This situation would be extremely rare in real scenarios.

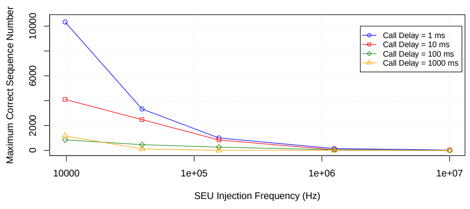

Fig. 6: Physical Hardware Results

5 Conclusion

The single event upset is among the most common types of system faults introduced by radiation, posing significant risk to spacecraft embedded systems.

Modern approaches to guarding against such faults typically introduce additional hardware to detect and correct SEU errors in target systems. In this paper, we present a software-only approach to protecting embedded system memory from SEUs, focused on the system stack. The stack is protected by injecting auxiliary assembly code within the target program. The prototype implementation is based on the AVR architecture, but is easily adapted to other architectures. Analytical and experimental results show that our approach detects and corrects SEU errors as expected.

6 Acknowledgments

This work is supported by the National Science Foundation through awards CNS-0745846 and CNS-1126344, and the City of Aiken, SC.

References

1. Celler, G. K., and Cristoloveanu, S. Frontiers of silicon-on-insulator. *Journal of Applied Physics 93*, 9 (2003), 4955–4978.
2. Czajkowski, D., and McCartha, M. Ultra low-power space computer leveraging embedded seu mitigation. In *IEEE Aerospace Conf* (2003), vol. 5, pp. 2315–2328.
3. Darling, P. Intel to Invest More than 5 Billion to Build New Factory in Arizona, October 2013. newsroom.intel.com/community/intel_newsroom/blog/2011/02/18/intel-to-invest-more-than-5-billion-to-build-new-factory-in-arizona.
4. Dutton, B. F., and Stroud, C. E. Single Event Upset Detection and Correction in Virtex-4 and Virtex-5 FPGAs. In *CATA* (09), W. L. 0025, Ed., ISCA, pp. 57–62.
5. Geluso, J. CRC16-CCITT, Jan 2014. srecord.sourceforge.net/crc16-ccitt.html.
6. Huang, L., and Selman, J. J. Watchdog timer, Dec. 2 1986. US Patent 4,627,060.
7. Irom, F., Farmanesh, F., Johnston, A., Swift, G., and Millward, D. Single-event upset in commercial silicon-on-insulator PowerPC microprocessors. *Nuclear Science, IEEE Transactions on 49*, 6 (2002), 3148–3155.
8. Lyons, R., and Vanderkulk, W. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM JRD 6*, 2 (1962), 200–209.
9. NASA. Space Shuttle Era Facts, October 2013.
10. NASA. JPL Institutional Coding Standard for the C Programming Language, April 2014. lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.
11. Oh, N., , et al. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability 51*, 1 (2002), 63–75.
12. OshonSoft. AVR SIMULATOR IDE, Jan 2014. www.oshonsoft.com/avr.html.
13. Shirvani, P., Saxena, N., and McCluskey, E. Software-implemented EDAC protection against SEUs. *Reliability, IEEE Transactions on 49*, 3 (2001), 273–284.
14. sourceware. Linker Scripts, 10 13. sourceware.org/binutils/docs/ld/Scripts.html.
15. ucsusa. USC Satellite Database, October 2013. www.ucsusa.org/nuclear_weapons_and_global_security/space_weapons/technical_issues/ucs-satellite-database.html.
16. Underwood, C., Ward, J., Dyer, C., and Sims, A. Observations of single-event upsets in non-hardened high-density srams in sun-synchronous orbit. *Nuclear Science, IEEE Transactions on 39*, 6 (1992), 1817–1827.
17. Underwood, C., Ward, J., Dyer, C., and Sims, A. Observations of single-event upsets in non-hardened high-density SRAMs in Sun-synchronous orbit. *Nuclear Science, IEEE Transactions on 39*, 6 (1992), 1817–1827.
18. Vincent L. Pisacane, H. K. U. Embedded Software Systems. In *Fundamentals of Space Systems Second Edition.* Oxford University Press, New York, 2005.