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

Author1, Author2, Author3, and Author4

Institute
`{author1,author2,author3,author4}@institute`

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

There are more than 1,000 operating satellites in orbit around Earth [15] today, and as of 2012, more than 130 manned spacecraft have been launched by the United States alone [9]. One of the most important factors that affect the reliability of such systems (and other equipment) is the reliability of the constituent embedded components which control the underlying telemetry systems, command systems, attitude control systems, and more [18]. Embedded software failures can cause serious consequences in this context.

The environment outside the Earth's atmosphere is highly radioactive and can cause damage to semiconductor devices [18]. One of the most common types of damage caused by radiation is the *single event upset* (SEU). Extremely small electronic components (i.e., tens of nanometers [3]) are used in modern integrated circuitry; the components cannot carry much charge. As a result, 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.

The main contributions of our work are as follows: (i) We present an approach that protects the system stack by injecting assembly code at the beginning and end of each application function. (ii) We present an implementation of the approach, using the popular AVR architecture as a target. (iii) We verify the protection efficacy of our approach and evaluate performance in terms of space and speed overhead using three applications with different stack usage patterns.

2   Related Work

Single event upset mitigation has been explored from both hardware and software perspectives. 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. They subject both types of microprocessors to proton and heavy-ion impacts, both known to cause SEUs. SEU rates were observably lower than those in commercial 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. Four software-based coding schemes are considered, comprising Hamming, Cyclic, Parity, and Reed-Solomon codes. 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.

Dutton and Stroud present a design implemented in configurable logic blocks for SEU detection and correction in the configuration memory of Field Programmable Gate Arrays (FPGAs) [4]. The architecture of the Xilinx Virtex FPGA is modified to implement an SEU controller that uses Hamming codes and parity values to detect and correct single bit errors in memory. This combination of Hamming and parity can also detect multiple bit upsets, but correction is not possible. The benefits include protection of the controller from SEUs and the high speed of error detection and correction, as compared to other methods.

3   Background

While our approach is architecture neutral, our implementation targets AVR microprocessors. Here we survey the AVR stack and function call process.

3.1   Stack Frame

The stack consists of stack frames, each corresponding to a function call. A stack frame is created when a function is called and freed when the function returns.
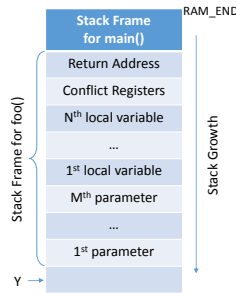

Fig. 1: AVR Stack Frame

For example, as shown in Figure 1, when the *main* function calls the *foo* function, a stack frame will be created for *foo*. First, the return address of *main* will be pushed to the stack, followed by the conflict registers. Next, the local variables and arguments will be pushed to the stack in reverse order of declaration. The stack frame spans the return address through the first argument. The stack frame pointer, Y, now points to the next available address in the stack. When *foo* finishes execution, the stack frame will be freed, and the stack frame pointer will point back to the position where the return address of the previous stack frame was stored.

3.2   Function Calls

All function calls follow the same process and use the system stack to perform most operations, as illustrated in Figure 2. Each rectangle represents two bytes

in the stack. SP denotes the stack pointer, and Y denotes the stack frame pointer. When a function is called, the return address is pushed onto the stack by one of the function call instructions, `call`, `rcall`, or `icall` (step 1). After the stack frame pointer is pushed (step 2), the stack frame of the function is created by changing the stack pointer and stack frame pointer (step 3). Arguments and locals are then pushed onto the stack (step 4), and the function begins executing (step 5). The arguments and locals are released after the function finishes its execution (step 6), and the stack frame pointer is restored (step 7). Finally, the function returns (step 8). The return address is popped and used when one of the return instructions, `ret` or `reti`, is called.
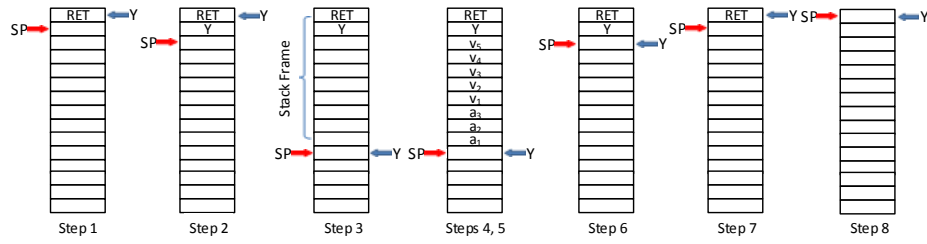

Fig. 2: Function Execution

4   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.

4.1 Supporting Memory Sections

To store stack frame replicas, two new sections are created in SRAM, as shown in Figure 3, just after the `.bss` section by modifying the linker script [14].

The `md` section is used to store stack frame replicas, which are referred to as *Stack Frame Snapshots* (SFSs). The heap section grows towards the stack. To prevent the `md` and `heap` sections from colliding, the size of the `md` section is fixed. If the heap is used, the size of the `md` section is set to 1/3 of the available space; otherwise, it is set to 1/2 of the available space. For example, if the `.data` and `.bss` sections require 1 KB in a 4 KB RAM, the space available is 3 KB, so the size of `md` is set to 1 KB. The `sp` section is used to store the address of the

Fig. 3: Modified Memory Sections

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.

4.2 Injected Code Segments

The injected code segments are designed to use only registers, reducing their dependency on RAM. Each segment is assigned a unique ID:

- *CRC Calculation* (ID: CC): Used to calculate the CRC of a memory region. In our implementation, CRC16-CCITT is used [5].
- *CRC Save* (ID: CS): Used to save the CRC to the stack.
- *CRC Compare* (ID: CM): Used to compare two CRCs. The comparison result indicates whether an SEU is detected.
- *Frame Copy* (ID: FC): Used to copy a stack frame to a given destination, and to save and restore stack frames.
- *Frame Size Save* (ID: FS): Used to save the size of the stack frame for the current function; this will be discussed in Section 4.3.
- *STP Initialization* (ID: SN): Used to initialize the STP so it points to the lowest address of the `md` section.
- *STP Update* (ID: SU): Used to update the STP. First, it obtains the correct STP value by comparing the three STP replicas. Next, the replicas are updated to reflect the addition or removal of a stack frame.

4.3 The ASM Handler

The ASM Handler is responsible for code injection. It consists of three loosely-coupled modules: the Reader, the Scanner, and the Injector.

Reader. The Reader is used to read the assembly file and generate corresponding metadata. Each metadata node classifies an assembly statement into one of three categories. A *directive* is used to specify assembly code information, such as the system architecture, section, and other information. A *label* is used to identify a location in the assembly code. An *instruction* is used to identify an instruction that will be executed by the microprocessor. The metadata also stores code injection information, which specifies whether code is to be injected before or after a given line, as well as the type of code to be injected. The Reader reads each line of assembly and generates a corresponding metadata node, which is then appended to a metadata list (in memory).

Scanner. The Scanner is used to scan the metadata list and mark each metadata node based on the operations performed by the associated code. Marked metadata nodes indicate that code segments will be injected either before or after the corresponding line of code.

We analyze the assembly code to identify the key operations where code segments must be injected — *stack frame establishment*, *stack frame pointer save*, and *function return*. The Scanner scans each node in the metadata list, checks if the code represented by the node performs one of the key operations and marks each such node with identifiers indicating the code segments to be injected and the position of the injection (i.e., before or after the assembly line).

The Scanner scans the metadata list to detect whether `malloc` is called in the target program, which indicates whether the `heap` section in RAM is used. This information is later used in determining the section size used to store the stack frame replicas, as discussed in Section 4.1. The Scanner also determines the size of each function's stack frame by scanning the assembly code used to establish the stack frame, "`sbiw r28, n`", yielding a stack frame size of $n + 10$. The `n` bytes are used to store the arguments and local variables, and the additional 10 bytes are used to store the return address, CRC, and three copies of the stack frame size, each of which requires 2 bytes.

Injector. The Injector is used to inject code segments into the target assembly code. It again scans the metadata list. When a node is marked, the Injector injects the specified code segments. Finally, a modified assembly file is generated, assembled, and linked to form an executable.

In our initial approach, the code segments were directly injected into the target code, effectively making the code segments *inlined*. The results showed that the ROM overhead was significant. Each function, regardless of its size, was injected with code segments that require approximately 500 bytes of ROM. In our final approach, all the code segments are injected at the end of the target code, and each is labeled with its unique ID. When scanning the metadata list, instead of injecting the code segments into the target code, a function call instruction, `call`, is injected. A function return instruction, `ret`, is added at the end of each

code segment. Because each segment was designed to use only registers, only two stack bytes are needed by each segment to save the return address. We discuss the performance of both approaches in Section 5.

4.4 Modified Function Execution Process

The auxiliary code modifies the function invocation and return processes.
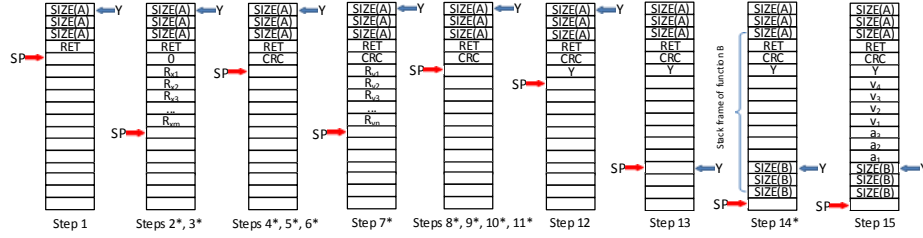

Fig. 4: Modified Function Invocation Process

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 4. 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 5. 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).
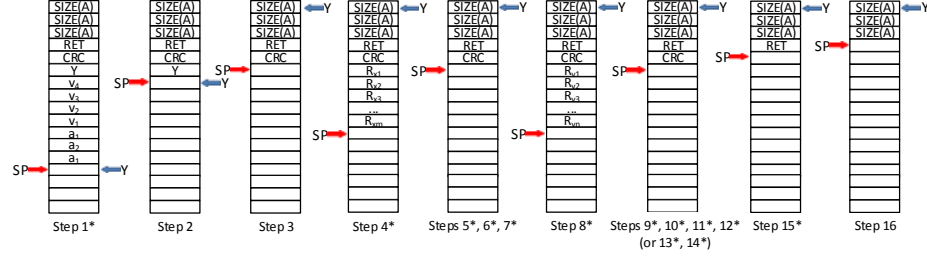

Fig. 5: Modified Function Return Process

5   Evaluation

We first introduce three test applications with different degrees of stack dynamism. We then validate the correctness of our approach and analyze the relationship between protection efficacy and SEU injection rate. Finally, we consider the overhead introduced, both in terms of space and execution speed.

5.1   Test Applications

To evaluate our approach under varying stack conditions and SEU injection rates, three AVR applications are considered. The stack usage pattern of
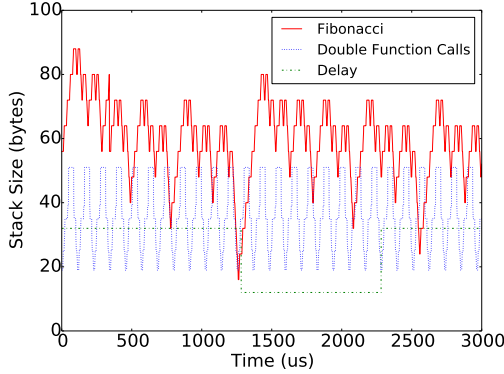

each application is shown in Figure 6. The x-axis represents execution time, and the y-axis represents stack size. Below is a description of each application. The `Delay` application repeatedly executes a function that contains a delay of 2,040 clock cycles, implemented using a while loop, yielding low stack variability. The `Double Function Calls` application repeatedly executes three functions — A calls B, and B calls C — yielding moderate stack variability. The `Fibonacci` application repeatedly calculates

Fig. 6: Stack Usage of Test Applications

the tenth Fibonacci number using recursion, yielding significant stack variability.

5.2   Validation

We first validate our approach and consider the SEU protection efficacy it affords. Recall the modified SRAM partition shown in Figure 3. 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. Again, the injected code segments used to protect the stack frames are designed to use only registers, and each segment requires only two bytes in the currently executing function's stack frame (for the return address).

We first assume that the currently executing function's frame, which includes the return address of the injected code segment, is not affected by SEUs. We use induction to prove the correctness of our approach. Suppose n is the number of stack frames stored in the stack, excluding the frame for *main*.

Base Case: If $n = 0$, only the stack frame of *main* is on the stack. When *main* calls another function, say *foo*, the stack frame for *foo* is created. According to our assumption, the current stack frame (*foo*'s) will not be affected by SEUs during execution. When *foo* returns, the stack frame of *main* is protected by our approach. So the stack frames of caller and callee are guaranteed to be correct if any function is invoked and returns when $n = 0$.

Induction Assumption: Assume that the stack frames of callers and callees are guaranteed to be correct for $n = k$, where $k \geq 1$.

Inductive Step: Now consider $n = k + 1$. Assume a is the current function, which calls b. According to our assumption, b's stack frame is not affected by SEUs. When b returns, the stack frame of a is protected by our approach. So the

stack frames of callers and callees are guaranteed to be correct when $n = k + 1$. The currently executing function's stack frame is assumed safe, and the stack frames of callers and callees are protected against SEUs during execution; by induction, the stack is guaranteed to be correct, assuming the current stack frame is never affected by SEUs.

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. 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:

$$p = 1 - \frac{c}{2s + e - c + 6} \tag{1}$$

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. Our approach succeeds when the following conditions are met: (i) the currently executing function's stack frame is not affected (so the return address of the injected code segment is not affected); (ii) at least two of the three copies of the caller's stack frame size are not affected; (iii) at least two of the three copies of the STPs are not affected; and (iv) at least one of the two caller's stack frames (the original and the backup copy saved in the md section) is not affected. To simplify the analysis, conditions (ii) and (iii) are strengthened, requiring that all three copies of the caller's frame size not be affected, and all three copies of the STPs not be affected. Since the strengthened conditions slightly reduce the probability of successful SEU protection (only 4 bytes are ignored), the real probability of protection is slightly higher than the presented results. The probability of successful protection can be expressed as:

$$\begin{aligned} p = &(1 - \frac{c}{2s + e - c + 6})^n * (1 - \frac{6}{2s + e - 2c + 6})^n \\ &* (1 - \frac{6}{2s + e - 2c})^n * \{(1 - \frac{2c}{2s + e - 2c - 6})^n \\ &+ C_2^1 * (1 - \frac{c}{2s + e - 2c - 6})^n * [1 - (1 - \frac{c}{2s + e - 3c - 6})^n]\} \end{aligned} \tag{2}$$

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 2, $(1 - \frac{c}{2s+e-c+6})^n$ is the probability that the currently executing function's stack frame is not affected by SEUs. $(1 - \frac{6}{2s+e-2c+6})^n * (1 - \frac{6}{2s+e-2c})^n$ is the probability that both copies of the caller's stack frame size and the three STPs are not affected by SEUs. Within the curly brackets, $(1 - \frac{2c}{2s+e-2c-6})^n$ is the probability that both the original and the copy of the caller's stack frame are not affected by SEUs. $C_2^1 * (1 - \frac{c}{2s+e-2c-6})^n * [1 - (1 - \frac{c}{2s+e-3c-6})^n]$ is the probability that either the original or the copy of the caller's stack frame are affected by SEUs. So $\{(1 - \frac{2c}{2s+e-2c-6})^n + C_2^1 * (1 - \frac{c}{2s+e-2c-6})^n * [1 - (1 - \frac{c}{2s+e-3c-6})^n]\}$ is the probability that at least one — the original or the copy — of the caller's stack frames is not affected.

In equation 2, the number of SEUs that occur, n, can be expressed as:

$$n = \frac{y * l}{m} * f \tag{3}$$

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.

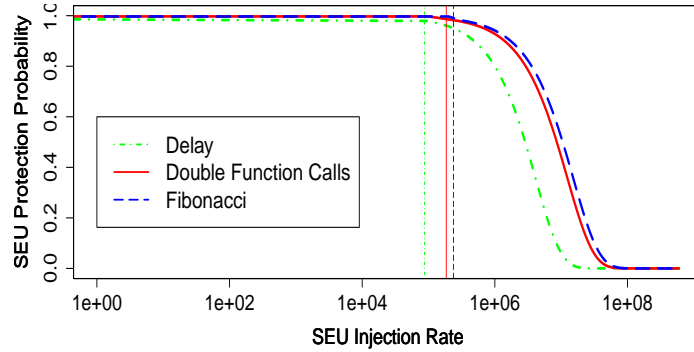| Applications	l	c	s	e
Fibonacci	42	10	60	2992
Double Function Calls	54	9	30	3022
Delay	115	16	30	3022

Table 1: Application Characteristics

Fig. 7: SEU Protection Probability

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, as shown in Table 1. Figure 7 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 because the SEUs may affect the stack frame of the current function, the stack frame sizes of the caller stored in the stack, the STPs, and stack frame copies stored in the `md` section. 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. Low stack frame dynamism causes the SEU protection probability for Delay to drop significantly compared to the other applications.

5.3 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. Table 2 summarizes the overhead of each injected code segment. The second column lists the number of times each code segment executes (per function execution), the third column lists the number of instructions executed in each code segment, the fourth column lists the number of clock cycles spent executing each code segment, and the fifth column lists the ROM space overhead for each injected segment. S denotes the size of the (recovered) stack frame. The *CRC calculation* 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. Therefore, 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.

Code Segment	Number of Execution	Instructions	Clock Cycles	ROM Space
CRC Calculation	2	24*S+1	27*S+4	50
CRC Save	1	13	26	26
CRC Compare	1	27	52	64
Frame Copy	1 or 2	64+4*S	8*S+128	50
Frame Size save	1	18	34	36
STP Initialization	1	16	28	32
STP Update	2	7	14	14
Total (No SEUs)	-	48*S+154	62*S+304	272
Total (Recovery)	-	52*S+218	70*S+432	272

Table 2: Execution Overhead

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 8.
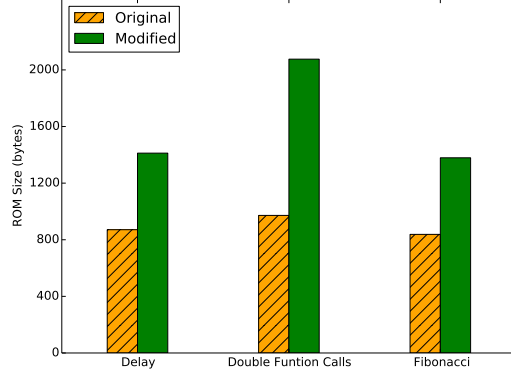

Fig. 8: ROM Overhead

The y-axis represents ROM size, in bytes. Delay and Fibonacci involve two functions, and Double Function Calls involves four. From Figure 8, 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.

We next evaluate execution overhead. As shown in Table 2, the execution overhead for every function call is determined by the size of the stack frame. We consider the execution overhead as a function of the average number of instructions executed between each call instruction (i.e., an inverse measure of call frequency). The execution overhead can be expressed as:

$$e = \frac{l + L}{l} \qquad (4)$$

Where L is the number of injected machine instructions for each function, and l is the average number of machine instructions executed between each call instruction. As summarized in Table 2, $L = 48 * S + 154$ when no SEUs are detected, and $L = 52 * S + 218$ when an SEU is detected. The average frame size, S, is 20. The overhead results are summarized in Figure 9. The x-axis represents the average number of instructions executed between function invocations (l), and the y-axis represents execution overhead, measured as the ratio between the execution speed of the original code and the modified code. The figure shows that given the same stack frame size, execution overhead is determined by stack dynamism. The less stack dynamism, the less speed overhead. The explanation is that within a given period of time, increased function calls lead to increased execution of the injected code. The results reveal an interesting tradeoff among dynamism, protection efficacy, and performance. Increasing dynamism offers better protection, but worse performance; decreasing dynamism offers better performance, but less protection. Knowledge of this tradeoff can be used to inform the function decomposition process, enabling embedded designers to appropriately balance protection efficacy and execution overhead.

5.4   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
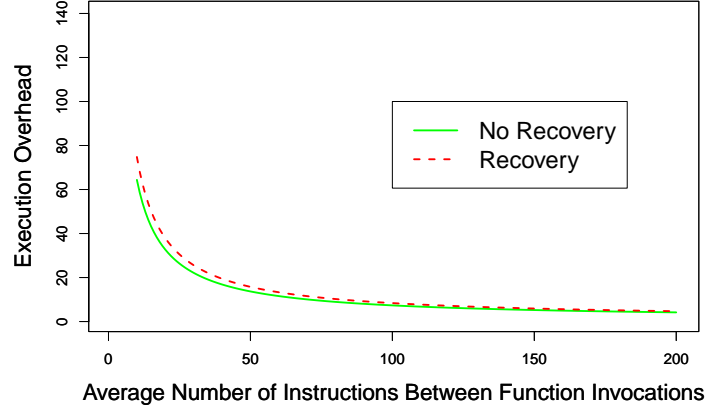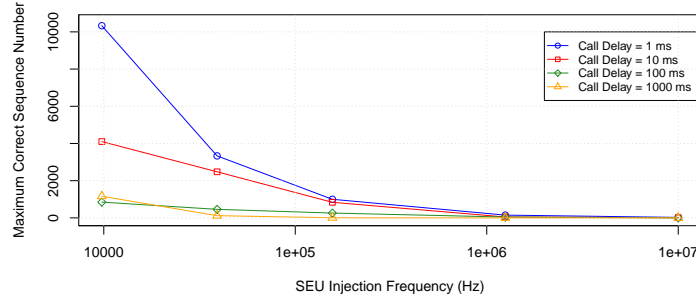
Fig. 9: Execution Overhead

Fig. 10: Physical Hardware Results

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. The bit flip frequencies are set to 10^7Hz, $1.25 * 10^6$Hz, $1.5625 * 10^5$Hz, 39062.5Hz, and 9765.625Hz[1]. An Atmega644 microprocessor is used in our experiments.

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 experimen-

---

[1] These frequencies are derived from the built-in timer prescaler of the Atmega644 at values of 1, 8, 64, 256, and 1024, respectively

tal results are summarized in Figure 10. 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.

6   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.

A study on protection efficacy has been provided, analyzing the probability of successful SEU protection as SEU frequency is increased. Experimental results for both ROM and execution (speed) overhead have been provided, using three applications exhibiting different degrees of stack dynamism. Since the size of the injected code is fixed, space overhead depends only on the number of functions in the target program and the size of each function. Speed overhead depends largely on function frame size and stack dynamism, as well as the occurrence rate of SEUs. Results show that for typical programs, our approach achieves a stack protection success rate of over 99%.

Future Work. Our future work follows four tracks. The first is focused on protecting global portions of RAM. In our current design, we ignore the `.data` and `.bss` sections used to store global and static variables. Since global and static variables are commonly used in embedded applications, introducing protection to these sections will improve the utility of our approach. The second involves introducing compression to the stack frame copy process. In our current design, stack frames are directly copied. Although compression will increase ROM and execution (speed) overhead, RAM usage and the probability that the stack frame copies are affected by SEUs can be reduced. Third, we plan to extend our approach to other GCC optimization levels. In the current design, optimization option -O0 is used. However, other options, such as -O2 or -Os, are frequently

used by developers. Applying our design to other levels involves studies on assembly code generated with other optimization levels and will further improve the utility of our approach. Finally, we plan to add support for external RAM. Internal RAM is a valuable resource in embedded systems. Our approach uses the internal RAM to store the stack frame copies, reducing the RAM space available for target programs. Adding external RAM can provide more space to store stack frame copies and offers more flexibility in future designs.

7   Acknowledgments

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.