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

Jiannan Zhai, Yangyang He, Fred S. Switzer, Jason O. Hallstrom
School of Computing
Clemson University
{*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. Our approach operates at the assembly level and injects assembly code into the target application to achieve memory protection without introducing additional hardware. Three applications are used to verify our approach, and the time and space overhead characteristics are evaluated.

1 Introduction

Humans have a longstanding curiosity about outer space. Since the launch of the first artificial satellite, Sputnik 1 [21], in 1957, over 6,000 satellites have been launched into space. There are more than 1,000 operating satellites in orbit around Earth [26] today, and an estimated 1,200 satellites will be launched over the next decade [24]. As of 2012, more than 130 manned spacecraft have been launched by the United States [17]. There are currently two operational space stations, and seven more are planned over the next decade [16] [6] [29] [25]. A total of 355 astronauts from 16 different countries have flown into space, among which 14 were killed during accidents [23]. Given the high cost and vital importance of spacecraft rovers and satellites, as well as their increasing functionality and complexity, the hardware and software reliability requirements are stringent.

One of the most important factors that affect the reliability of spacecraft (and other equipment) is the quality of the constituent embedded components which control telemetry systems, command systems, attitude control systems, and more [28, p.654]. For example, the MSX (Midcourse Space Experiment) spacecraft, launched in the mid-1980s, was equipped with 54 embedded processors, running more than 275,000 lines of code, managing 19 subsystems [28, p.655]. Embedded software failures could cause serious consequences in this context. In 1996, the Ariane 5 spacecraft, which took 10 years and 7 billion (US) dollars to build, crashed due to the failure of the Flight Control Subsystem when it performed a conversion from a 64-bit floating point value to a 16-bit signed integer value [13].

The environment outside the Earth's atmosphere is highly radioactive. The radiation is mostly generated by the sun and other stars, and can cause damage to semiconductor devices [28, p.636]. 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 [7]) 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 could 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 [28, p.637]). The damage caused by an SEU can range from system dysfunction to system crash, both of which are intolerable for spacecraft embedded systems.

Modern approaches used to prevent and correct SEU errors often introduce additional hardware and software to the target system. Systems designed to be used in space often contain processors fabricated using Silicon On Insulator technology, which effectively reduces the size of the processor, and therefore reduces the vulnerable area that a highly-charged particle could strike. Triple modular redundancy (TMR) [28, p.645] uses three identical systems (either physically or temporally duplicated) to simultaneously perform identical tasks, where the correct result is then decided by a majority vote. Due to the inherent randomness of Single Event Upsets, the chance that two charged particles would strike parallel locations in two different systems at the same time is incredibly low, making TMR a very popular practice in spite of the memory expense. Another popular SEU correction method is the use of codes such as the Hamming Code to correct bit errors [20]. However, most of these processes are supplemented by periodic memory scrubbing and hardware watchdog timers used to check execution flow and abnormal program executions caused by SEUs [28, p.648].

In this paper, we present an 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 customized code into the target assembly generated by AVR-GCC. After injection, each callee computes and saves the checksum of its caller's current stack frame, and duplicates the caller's stack frame when the callee enters its function body. Before the callee returns, it verifies the stack frame of the caller using the saved checksum, and overwrites the stack frame using the duplicate, if an SEU is detected. Our approach changes the target system software and does not introduce additional hardware. Since our approach operates at the assembly level, it is language and application neutral. To demonstrate our approach, an AVR microprocessor, the ATmega644 [2], is used in the paper.

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 function, which calculates and saves the CRC of a given function's stack frame, saves the stack frame, and restores the stack frame if the newly calculated CRC does not match the saved CRC. (ii) We implement a tool used to scan the assembly code generated from the target program, and to inject auxiliary assembly code into the target program. (iii) We verify the protection efficacy of our approach, and evaluate the performance in terms of space and speed overheads using three applications with different stack usage patterns.

Paper Organization. Section 2 summarizes key elements of related work. Section 3 provides background content related to our approach, including the microprocessor architecture, AVR function call process, and AVR toolchain. Section 4 presents the design and implementation of our approach. Section 5 presents an evaluation of the approach, with an emphasis on ROM size and speed overhead. Finally, Section 6 concludes with a summary of contributions and pointers to future work.

2 Related Work

Prior work on single event upset mitigation spans two categories: First, robustness at the device-level may be improved by building the hardware in such a way that radiation does not cause upset events. Common hardware modifications include physical shields, protective insulation, and error correcting latches. Second, design-level robustness may be improved by incorporating libraries and software designed to protect from single event upsets through the use of Hamming or other codes, or triple modular redundancy.

2.1 Device-Level Robustness

One of the primary methods of device-level radiation hardening is to fabricate processors using Silicon on Insulator (SOI) technology. In this process, transistors are placed upon a thin layer of silicon, which is then placed on top of an insulator, reducing the capacitance of the switches, and the size of processors [3]. Reducing processor size effectively reduces the area over which highly-charged particles can strike, statistically reducing impacts, and therefore errors.

Irom et al. [12] compare SEU error rates in SOI microprocessors to conventional microprocessors. They subject both types of microprocessors to proton impacts within a cyclotron, and to heavy-ion impacts within an accelerator, both of which are known to cause SEUs in processors. From these tests, Irom et. al. conclude that due to the significant reduction of cross sections in SOI microprocessors, SEU rates are lower than those in commercial microprocessors.

She et al. [19] improve the design of conventional latches by implementing an error detection circuit and a multiplexer. While conventional latches are susceptible to voltage changes caused by SEUs, the proposed latch uses an error detection circuit that checks for faults using the precharge and discharge operations. This latch then uses a multiplexer to output a corrected signal based on the fault detected by the error detection circuit. During comparison tests, the authors found that the proposed latch introduced little overhead and offered good performance, as well as better SEU protection than the conventional latch.

2.2 Design-Level Robustness

The most popular methods of radiation hardening are at the design level, forgoing the need to modify the hardware, thereby allowing the use of commercial microcontrollers. Some of these methods involve parity bits and linear error-correcting codes [9], such as the Hamming Code, which correct bit errors by storing extra information about data in larger blocks. Alternately, Triple Modular Redundancy (TMR) [14] is a voting-based approach that prevents errors by implementing three systems, and then using the common result. The motivation is that the likelihood of two single event upsets occurring at precisely the same time, in the same location, in two different devices, is incredibly low, making TMR popular in spite of the added expense. While this method can be implemented via hardware (such as in RAID [5]), it is also effective as a software-only strategy.

Shirvani et al. [20] examine a set of Error Detection and Correction (EDAC) methods. These methods detect and subsequently correct errors in memory, such as those caused by SEUs. EDAC methods come in both hardware and software forms. In scenarios where hardware-based approaches are cost prohibitive, software-based methods work well. The authors found that the reliability of software-based methods tends to decrease over time more rapidly than hardware-based methods. However, the rate of reliability loss is low enough to still be more cost efficient than hardware-based methods. Four software-based coding schemes were considered, comprising Hamming, Cyclic, Parity, and Reed-Solomon codes. The authors found that most EDAC implementations can be improved by periodically scrubbing (completely cleaning) memory.

Mhatre and Aras present a design [15] for the on-board computer of the COEP Student Satellite, a Ham communications pico-satellite. SEU protection on the satellite involves implementation of Hamming codes, Triple Modular Redundancy, and watch-dog software. In the Hamming code, each 32-bit instruction is coded in the form of a 38-bit codeword, where the redundant bits are used for parity. Single bit errors are corrected by comparing these parity values with pre-calculated values. Triple Modular Redundancy is used in storing and protecting these parity values, saving space by not implementing TMR over the entire instruction memory. This extra storage increases program memory requirements by 75%, rather than the 200% increase required by a full implementation of TMR. The watch-dog software prevents other errors by automatically escaping system crashes.

Similarly, 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) [8]. 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 still not possible. The benefits include the protection of the controller from SEUs and the high speed of error detection and correction as compared to other methods.

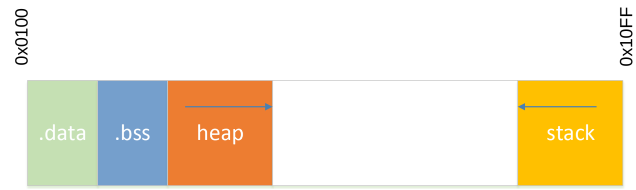


Figure 1: AVR RAM Map

3 Background

While our approach is architecture neutral, our implementation is based on the Atmel AVR toolchain and focuses on AVR microprocessors. In this section, we present the information on the AVR architecture, function calls, and using AVR Toolchain.

3.1 AVR Architecture

AVR microprocessors are based on a modified Harvard architecture [1], which stores instructions and data in physically separate memories, flash memory and SRAM, respectively. It concurrently accesses instructions and data through separate memory buses. Flash memory is non-volatile and offers high capacity, but slow access speed, used to store executable programs composed of AVR instructions. The SRAM is volatile and offers low capacity, but fast access speed, used to store data used by the executable programs at runtime. The ATmega644 includes 64KB of flash memory, 4KB of SRAM, a 16-bit instruction bus, and an 8-bit data bus.

3.1.1 AVR SRAM

The on-board SRAM of the ATmega644 has an address range of 0x0100 to 0x10FF, as shown in Figure 1. It can be extended to address 0xFFFF using external RAM. The SRAM is partitioned into sections, each used to store different types of data. The *.data* section is used to store initialized static variables and global variables. The *.bss* section is used to store uninitialized static and global variables. The Pre-allocated SRAM usage is the sum of the sizes of the .data and .bss sections. The remaining space in SRAM is shared by the heap and stack sections. The *heap* section is used to store dynamically allocated memory, e.g. when *malloc()* is called [10]. The heap grows “upward”, towards the higher address range. The *stack* section is used to store return address, actual parameters, conflict registers, local variables, and other information. The stack grows “downward”, from the *RAM_END*, address 0x10FF, towards the lower address range.

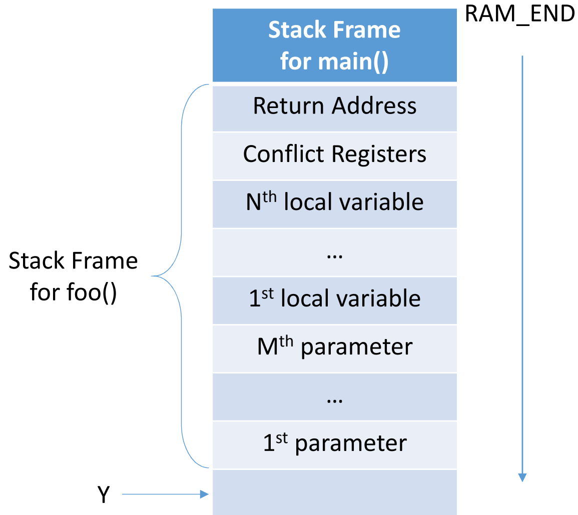


Figure 2: AVR Stack Frame

3.1.2 Stack Frame

The stack consists of stack frames, each corresponding to a function call. Each stack frame is created when a function is called and freed when the function returns. For example, as shown in Figure 2, when the *main* function calls the *foo* function, a stack frame will be created for the *foo* function call. First, the return address of the *main* function will be pushed to the stack, followed by the conflict registers. Next, the local variables and parameters will be pushed to the stack in reversed order of their declarations. The stack frame spans the return address through the first parameter. The stack frame pointer 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.

3.1.3 Registers

AVR microprocessors have two types of different registers, general-purpose registers, and I/O registers.

General-purpose registers are used for arithmetic operations, such as adding, subtracting, and comparing numbers, as well as indexing and setting long jump destinations. The ATmega644 has 32 general-purpose registers, R1 through R32, which are mapped into the first 32 locations of the RAM space, and can be directly used in assembly commands. Some general-purpose registers are used for special purposes; for example, R29 and R28 store a 16-bit address, the Y-Pointer, to indicate the top of the current stack frame. (The use of the Y pointer will be explained in the next subsection). The use of these registers is compiler-dependent. For example, AVR-GCC uses R24 and R25 to store the return value of each function call. We attempt to limit the number of registers manipulated by our approach to reduce the cost of saving and restoring the conflict registers.

The I/O registers are used to control the internal peripherals of the AVR microprocessor. The ATmega644 has 64 I/O Registers, mapped into the next 64 locations of the SRAM space, *0x20* through *0x5F*. Again, some I/O registers are used for special purposes; for example, AVR-GCC uses *0x3E* and *0x3D* as the stack pointer (SP), which indicates the current top of the stack.

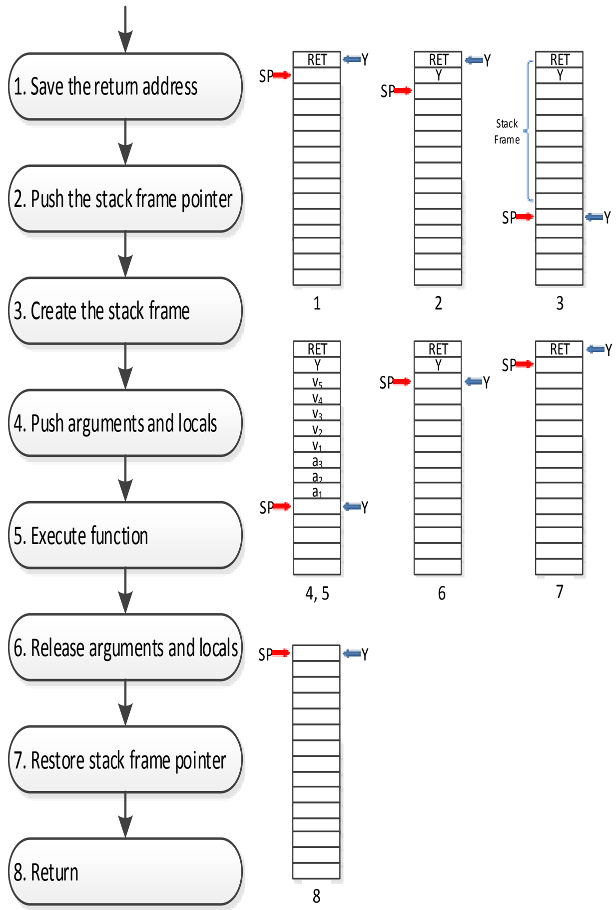
3.2 Function Calls

All function calls follow the same process and use the system stack to perform most operations, as illustrated in Figure 3. Figure 3a explains the execution process when a function is called, and Figure 3b shows the associated stack changes after each operation is performed. Each rectangle box represents two bytes in the stack. The numbers below each stack denote the operation(s) that changed the stack. `SP` denotes the stack pointer, and `Y` denotes the stack frame pointer. When a function is called, the return address is automatically 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). The arguments and local variables are then pushed onto the stack (step 4), and the function begins executing (step 5). The arguments and local variables 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 function return instructions, `ret` or `reti`, is called.

3.3 Atmel AVR Toolchain

The Atmel AVR toolchain is a collection of tools used to generate executable programs for AVR microprocessors. The toolchain consists of the following tools.

- *avr-gcc*, an extension of GNU GCC, is a cross compiler which translates high-level C or C++ code to assembly code for AVR microprocessors.
- *avr-as* is an assembler, which translates AVR assembly code to an object file.
- *avr-ld* is a linker, which uses a linker script to combine object modules into an executable image suitable for loading into the flash memory of an AVR microprocessor. By using a customized linker script, the default locations and sizes in the SRAM



(a) Process (b) Stack

Figure 3: Function Execution

can be changed. Further, new memory sections can be inserted into the SRAM [4], a facility used by our approach.

- *avr-libc* is a standard C library, which contains standard C routines as well as many additional AVR-specific library functions.

As a matter of convenience, the AVR toolchain can be used to compile, assemble, and link C programs in a single command. However, in order to modify the assembly code of an AVR application and use customized linker script, these steps are performed done individually.

AVR GCC provides 5 optimization levels, -O0, -O1, -O2, -O3, and -Os, each providing different optimization options. The exception is -O0, which offers no optimization [11]. Our approach is based on modifying unoptimized assembly code generated with the -O0 option. The reason is without optimization, generated assembly code will show the original intention of each line in the source program, and the operations to stack will be more

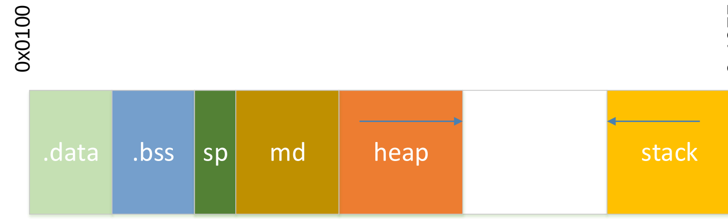


Figure 4: Modified Memory Sections

meaningful. Further, this option makes it more convenient for developing, debugging, and evaluating our approach. However, to improve the applicability of our approach, we plan to extend our approach to other optimization levels in our future work.

4 System Design/Implementation

We focus on issues surrounding the protection of the runtime stack, and therefore we make the following assumptions: i) Flash memory and registers are not affected by SEUs. ii) Only one SEU will occur during a given function execution. This assumption is well-justified. It is rare for more than one bit to be upset simultaneously; this occurs in only 5 to 6 percent of bit flip errors [27].

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 and handles CRC calculation, CRC comparison, and memory duplication. 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 of the function 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.

The ASM Handler, written in Java, performs the code injection, as illustrated in Figure 5. First, the input C code is compiled to assembly using GCC. The ASM Handler then injects the assembly code. Finally, the modified code is assembled and linked into an AVR executable. In this section, we discuss the supporting memory sections, the code segments injected into the target program, the architecture of the ASM Handler, and the function execution process after code injection.

4.1 Supporting Memory Sections

To store the duplicate stack frames, two new sections are created in SRAM as shown in Figure 4, just after the `.bss` section by modifying the linker script [22].

The `md` section is used to store duplicate stack frames. These duplicate frames are referred to as *Stack Frame*

Snapshots (SFSs). The heap section grows towards the stack, and of course, the runtime usage of the heap and the stack are unpredictable. 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 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 duplicate could be altered by a flipped bit. The altered STP is easily identified and corrected by comparing the values of the three STP duplicates.

4.2 Injected Code Segments

We categorize the injected code based on function. Each continuous assembly segment performs a set of operations, handling a specific action. These segments are designed to use only registers, reducing their dependency on RAM. Each segment is assigned a unique ID. Here we summarize each type of code segment.

- The *CRC Calculation* segment (ID: CC) is used to calculate the CRC checksum of a given memory region, e.g., a stack frame. In our implementation, CRC16-CCITT is used.
- The *CRC Save* segment (ID: CS) is used to save the CRC checksum to the stack.
- The *CRC Compare* segment (ID: CM) is used to compare two CRC checksum. The comparison result indicates whether an SEU is detected.
- The *Frame Copy* segment (ID: FC) is used to copy a stack frame to a given destination, and to save and restore stack frames.
- The *Frame Size Save* segment (ID: FS) is used to save the size of the stack frame for the current function; this will be discussed in Section 4.3.3.
- The *STP Initialization* segment (ID: SN) is used to initialize the STP so it points to the lowest address of the `md` section.
- The *STP Update* segment (ID: SU) is used to update the STP. First, it obtains the correct STP value by comparing the three copies of the STP. Next, all three copies are updated. The segment increases the STP to save a stack frame in the SFS and decreases the STP to release a stack frame from the SFS.

```
1 .arch atmega644              % directive
2    .text                     % directive
3 .global main                 % directive
4    .type main, @function     % directive
5 main:                        % label
6    push r28                  % instruction
7    push r29                  % instruction
8    ...
```

Listing 1: Assembly Code Example

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, as shown in Figure 5. Assembly metadata is generated to assist the code injection process. Here we describe the metadata creation process and describe each module of the ASM Handler.

4.3.1 ASM Metadata

Each line of assembly code is tagged with metadata representing the line of code. The metadata annotation classifies each line into one of three categories, as shown in Listing 1. A *directive* is used to specify assembly code information, such as the system architecture (line 1) and section (line 2), a label (line 3), the label type (line 4), and other information. A *label* is used to identify a location in the assembly code (line 5). In this example, `main` specifies the starting address of the main function. An *instruction* is used to identify an instruction that will be executed by the microprocessor (lines 6-8). The metadata also stores code injection information, which specifies whether code is injected before or after a given line, as well as the type of code to be injected.

4.3.2 Reader

The Reader is used to read the assembly file and generate corresponding metadata. It reads each line of assembly and generates a corresponding metadata node (in memory). The metadata node is then appended to a metadata list. For example, the Reader generates a list with 7 nodes after it reads the assembly code in Listing 1, as shown in Figure 5.

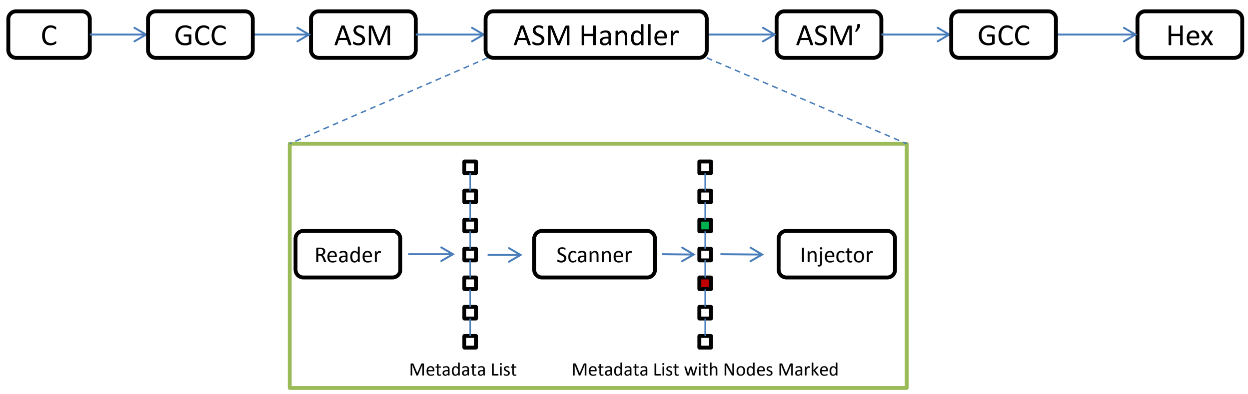


Figure 5: Code Injection Process

4.3.3 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. Here we summarize the key operations.

- The *Stack Frame Establishment* operation is used to establish the stack frame for the current function. This operation is identified by scanning "`sbiw r28,`n", which is used to establish the stack frame.

- The `Stack Frame Pointer Save` operation is used to copy the stack frame pointer, `Y`, to the stack pointer, `SP`. This operation is identified by scanning "`out __SP_L__,r28`", which indicates that the required registers are ready, and the function is about to execute.

- The `Function Return` operation is used when a function returns. This operation is identified by scanning the return instruction, "`ret`".

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 a list of identifiers for the set (CC, CS, CM, FC, FS, SN, SU, P), where CC, CS, CM, FC, FS, SN, SU are the IDs of the code segments to be injected, and P indicates the position of the injection (i.e., before or after the assembly line). Nodes that do not require code injection are not tagged. For example, the metadata node that represents the Function Return operation is marked with (CC, CM, FC, SU, P), indicating code segments CC, CM, FC, and SU must be injected before the associated line of code.

The Scanner also extracts two information elements from the metadata list. i) The Scanner scans the metadata list and detects if `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 duplicates, as discussed in Section 4.1. ii) The Scanner 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 stack frame size, each of which takes 2 bytes.

4.3.4 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 in the position specified by parameter *P*. Finally, a modified assembly file is generated, which is then assembled and linked to form an executable file.

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`, iss injected. A function return instruction, `ret`, is added at the end of each code segment. Because each segment was designed to use only registers, two bytes are needed by each segment to save the return address. To protect the return address, two additional copies of the return address are pushed onto the stack at the beginning of each code segment. the correct return address is retrieved by comparing the three duplicates at the end of each segment. We discuss the performance of both approaches in Section 5.

4.4 Modified Function Execution Process

The auxiliary code injected at the beginning and end of each function modifies function execution process, including function invocation process and function return process. Below is a description of both modified processes.

4.4.1 Modified Function Invocation Process

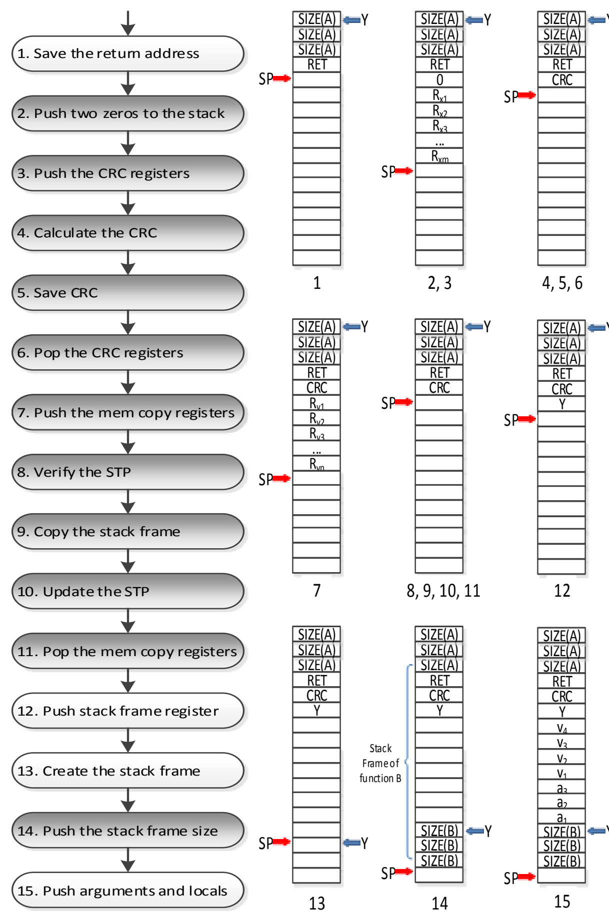
The code segments injected at the beginning of each function are used to calculate a CRC over the caller's stack frame, and to save a duplicate of the caller's stack frame, as shown in Figure 6. Figure 6a shows the execution process of the pre-invocation code; Figure 6b shows the stack changes associated with the pre-invocation code. Each rectangle box represents two bytes in the stack. In the execution process diagram, the white ovals show the operations performed by the original code, and the shaded ovals show the operations performed by the injected code. In the stack diagram, `SP` denotes the stack pointer, and `Y` denotes the stack frame pointer. As before, the numbers below each stack identify the operations that changed the stack.

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 (zeros) 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 stack frame are pushed onto the stack (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 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 stack frame size of the callee, function `B`, are pushed onto the stack (step 14), which is a key operation in the injected code.

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

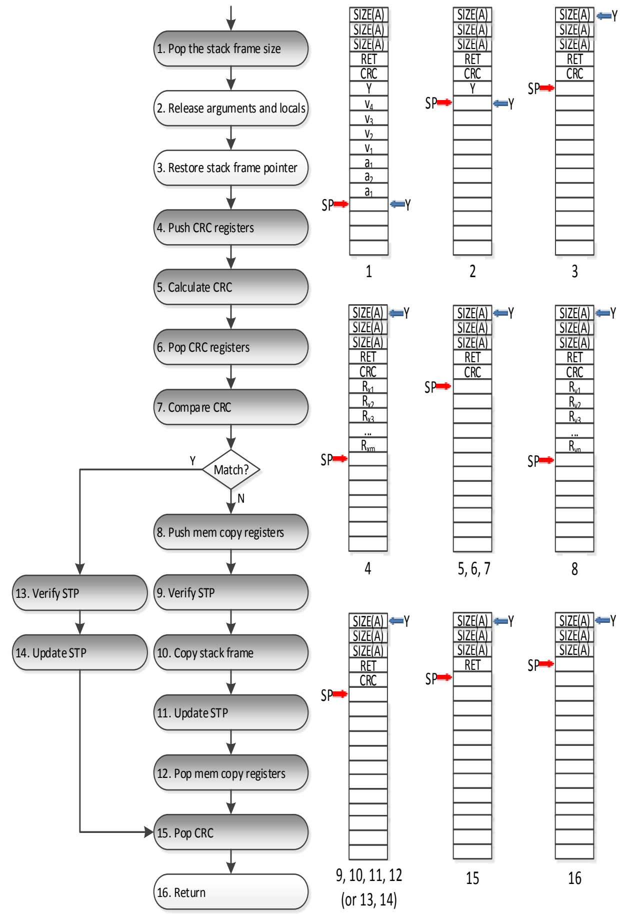


(a) Process (b) Stack

Figure 6: Modified Function Invocation Process

4.4.2 Modified Function Return Process

The code segments injected at the end of each function are used to verify the stack frame of the caller function, and to restore the stack frame if an SEU is detected, as shown in Figure 7. Figure 7a shows the execution process of the post-invocation code; Figure 7b shows the stack changes associated with the post-invocation code. Each rectangle box represents two bytes in the stack. Again, in the execution process diagram, the white ovals show the operations performed by the original code, and the shaded ovals show the operations performed by the injected code. Other diagramatic conventions mirror previous figures.



(a) Process Post (b) Stack Post

Figure 7: Modified Function Return Process

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 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).

5 Evaluation

We now present the results of our experimental analysis to evaluate the SEU protection approach. We first introduce three test applications with different degrees of stack dynamism to evaluate the protection efficacy of our approach. We then validate the correctness of our approach, and analyze the relationship between the successful SEU protection probability and the SEU injection rate. Finally, we consider the overhead introduced by our approach, both interms of space and execution speed. Ubuntu 13.10, with Linux kernel version 3.8, and GCC 4.1.2 are used throughout.

5.1 Test Applications

To evaluate our approach under varying stack conditions and SEU frequencies, three AVR applications are introduced. The stack usage pattern of each application is shown in Figure 8. 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 4400 clock cycles, implemented using a while loop, yielding low stack variability.
- The **Double Function Calls** application repeatedly executes three functions — function A calls B, and function B calls C — yielding moderate stack variability.
- The **Fibonacci** application repeatedly calculates the tenth Fibonacci number using recursion, yielding significant stack variability.

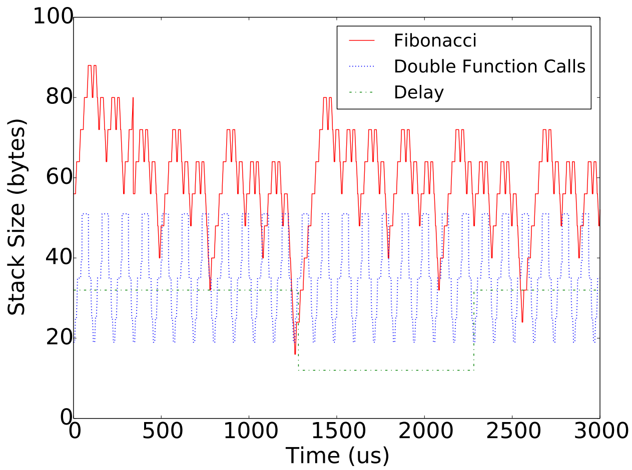


Figure 8: Stack Usage of Test Applications

5.2 Validation

We first validate our approach and consider the SEU protection efficacy it affords. We focus our analysis on stack frame protection.

We first assume that the stack frame of the currently-executing function is not affected by SEUs. We use induction to prove the correctness of our approach. Suppose the current stack contains n stack frames. If $n = 1$, the currently-executing function is the *main* function. It will not be affected according to the assumption. Assume the stack is protected when $n = k \ (k \geq 2)$. Now consider when $n = k-1$. Since the currently-executing function's stack frame, the $(k-1)$th stack frame, is protected by it's callee, when the callee returns, the $(k-1)$th stack frame is guaranteed to be correct. Since the current stack frame is not affected according to the assumption, the stack is protected when $n = k-1 \ (n \geq 2)$. By induction, the stack is protected when the current stack frame is not affected.

To verify this claim, the AVR Simulator IDE [18] was used to manually inject SEUs and observe the execution results. The results show 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:

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

Where p is the probability of successful protection, c is the size of the current stack frame, s is the size of the stack, and e is the size of the unused space between the heap and the stack.

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

Table 1: Applications Stack Characteristic

We extend our analysis to cases where more than one SEU may occur during a given function execution. The probability of successful SEU protection can be expressed as:

$$\begin{aligned} p = & (1-\frac{c}{s+e})^n\{(1-\frac{6}{s+e-c})^n \\ & +C_3^2(1-\frac{4}{s+e-c})^n[1-(1-\frac{2}{s+e-c-4})^n]\} \end{aligned} \tag{2}$$

Where p is the probability of success, c is the size of the current stack frame, s is the size of the stack, e is the size of the unused space between the heap and the stack, and n is the number of SEUs that occur during the function's execution. Our approach succeeds when the currently-executing function's stack frame is not affected and at most one of three copies of the caller's stack frame size is affected. In equation 2, $(1-\frac{c}{s+e})^n$ is the probability that the currently-executing function's stack frame is not affected. $(1-\frac{6}{s+e-c})^n$ is the probability that all the three copies of the caller's stack frame are not affected, when current stack frame is not affected. $C_3^2(1-\frac{4}{s+e-c})^n[1-(1-\frac{2}{s+e-c-4})^n]$ is the probability that any two of the three copies are not affected regardless of the other one, when current stack frame is not affected. The component within the braces is the probability that at most one copy is affected, when current stack frame 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 an instruction, m is the frequency of the microprocessor, l is the number of instructions in the current function, and f is the rate at which SEUs are injected in RAM. In our configuration, most AVR instructions cost 2 clock cycles to execute, and the frequency of our ATmega644 is set to 10MHz.

We now consider the relationship between the successful SEU protection probability and the SEU occurrence rate. To demonstrate the relationship, we collect the corresponding parameters of the three test applications using AVR Simulator IDE, as shown in Table 1. Figure 9

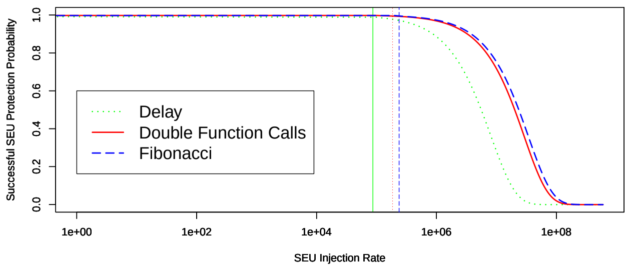


Figure 9: Successful SEU Protection Probability

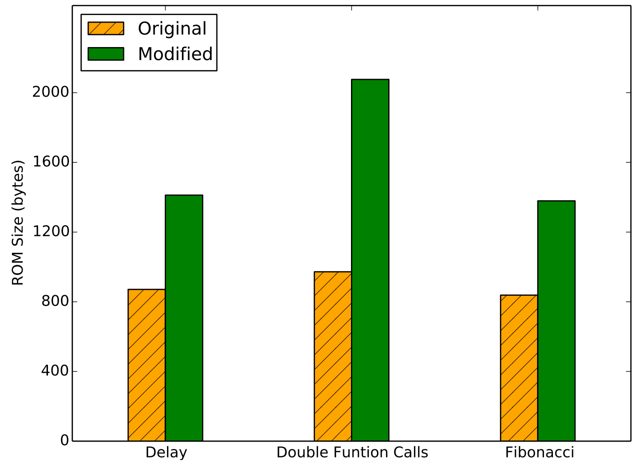


Figure 10: Space Overhead

plots the change in successful SEU protection probability as a function of SEU injection frequency. The x-axis denotes the rate at which SEUs are injected, and the y-axis denotes successful SEU protection probability. Each vertical line shows where the number of SEUs begins to exceed 1 (for each application). For each application, when only one SEU occurs during a given function execution (left side of the vertical line), the successful SEU protection probability is constant (Delay: 99.48%, Double Function Calls: 99.71%, Fibonacci: 99.67%), 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 successful SEU protection probability increases because the SEUs may affect both the stack frame of the current function and the stack frame size of the caller stored in the stack. As the SEU occurrence rate increases, the successful 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 successful SEU protection probability of Delay drop significantly before the other two applications, as shown in Figure 9.

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), 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. Therefore, the minimum overhead introduced in terms of number of clock cycles is $62*S+304$, when no SEU is 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 10. The y-axis denotes the ROM space overhead, in bytes. Delay and Fibonacci involve two functions, and Double Function Calls involves four. From Figure 10, we can see that the ROM overhead of the Double Function Calls application is twice the Delay and Fibonacci applications. The ROM overhead is related only to the number of functions in the program. We next evaluate execution speed overhead. The execution time is obtained using AVR Simulator IDE. The results are summarized in Figure 11. The y-axis represents the execution time, obtained using AVR Simulator IDE. Each *Original* bar shows the execution time of the original application; each *No Recovery* bar shows the execution time of the protected application with no SEUs detected; each *Recovery* bar shows the execution time of the protected application with SEUs detected. The figure shows that the stack variability impacts the speed overhead. The greater the stack dynamism, the greater the speed overhead. The explanation is that high stack dynamism shows dense function invocation, and indicates high execution speed overhead caused by the injected code.

6 Conclusion

The Single Event Upset is among the most common types of faults introduced by radiation, posing significant risk to spacecraft embedded systems. Modern approaches to guarding against such faults often introduce

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 Recovery)	-	48*S+154	62*S+304	272
Total (Recovery)	-	52*S+218	70*S+432	272

Table 2: Speed Overhead

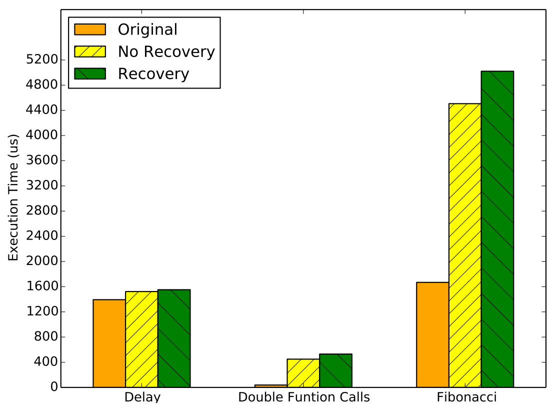


Figure 11: Speed Overhead

additional hardware and software 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.

Our approach focuses on the system stack, which is the most important and dynamic region in memory. 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. Experiment 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). Performance evaluation of both ROM and execution (speed) overhead is provided, using three applications with 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 stack protection success rate of over 99%.

Future Work. Our future work includes three components. The first 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. Second, we plan to extend our approach to other GCC optimization levels. In the current design, optimization level option -O0 is used. However, other options, such as -O2 or level -Os, are usually used. Applying our design to other levels involves studies on assembly code generated with other optimization levels, and will greatly improve the adaptability of our approach. Third, we plan to study scenarios where SEU occurrence is not uniformly distributed. In this paper, a uniform distribution is assumed. However, in real systems, the distribution depends on the altitude and angle of the device towards the sun. Finally, we plan to add 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 the target program. Adding external RAM can provide more space to store stack frame copies and offer more flexibility in future designs.

7 Acknowledgments

This work is supported by the National Science Foundation through awards CNS-0745846 and CNS-1126344.

References

[1] ARGADE, P. V., AND BETKER, M. R. Apparatus and method for computer processing using an enhanced harvard architecture utilizing dual memory buses and the arbitration for data/instruction fetch, Nov. 19 1996. US Patent 5,577,230.

[2] ATMEL. Atmel avr 8-bit and 32-bit microcontrollers, October 2013. www.atmel.com/products/microcontrollers/avr/default.aspx.

[3] CELLER, G. K., AND CRISTOLOVEANU, S. Frontiers of silicon-on-insulator. *Journal of Applied Physics 93*, 9 (2003), 4955–4978.

[4] CHAMBERLAIN. Using ld, January 1994. ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html.

[5] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June 1994), 145–185.

[6] COPPINGER, R. Us, russian capsules vie for orbital domination, October 2013. www.space.com/8968-piece-chinese-space-station-assembled-2011-launch.html.

[7] 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.

[8] DUTTON, B. F., AND STROUD, C. E. Single event upset detection and correction in virtex-4 and virtex-5 fpgas. In *CATA* (2009), W. L. 0025, Ed., ISCA, pp. 57–62.

[9] ELIAS, P. Error-free coding. *Information Theory, Transactions of the IRE Professional Group on 4*, 4 (1954), 29–37.

[10] GOLDT, S., VAN DER MEER, S., BURKETT, S., AND WELSH, M. The linux programmer's guide. *Linux Documentation Project (ftp://sunsite. unc. edu/pub/Linux/docs/linux-doc-project/programmers-guide/llpg-0.4. tar. gz)* (1995).

[11] HOSTE, K., AND EECKHOUT, L. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), ACM, pp. 165–174.

[12] 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.

[13] LIONS, J. L. Ariane 5 flight 501 failure report, October 2013. www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[14] LYONS, R., AND VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development 6*, 2 (1962), 200–209.

[15] MHATRE, N., AND ARAS, S. A hybrid approach to radiation fault tolerance in small satellite applications. In *Proc. 62^{nd} International Astronautical Congress* (2011).

[16] MOKOWITZ, C. First piece of chinese space station assembled for 2011 launch, October 2013. www.space.com/8968-piece-chinese-space-station-assembled-2011-launch.html.

[17] NASA. Space shuttle era facts, October 2013.

[18] OSHONSOFT. Avr simulator ide, January 2014. www.oshonsoft.com/avr.html.

[19] SHE, X., LI, N., AND TONG, J. Seu tolerant latch based on error detection. *Nuclear Science, IEEE Transactions on 59*, 1 (2012), 211–214.

[20] SHIRVANI, P., SAXENA, N., AND MCCLUSKEY, E. Software-implemented edac protection against seus. *Reliability, IEEE Transactions on 49*, 3 (2001), 273–284.

[21] SIDDIQI, A. A. In *Sputnik and the Soviet Space Challenge*. University Press of Florida, Gainesville FL, 2003, p. 155.

[22] SOURCEWARE. Linker scripts, October 2013. sourceware.org/binutils/docs/ld/Scripts.html.

[23] SPACE.COM. Nasa's space shuttle by the numbers, October 2013. www.space.com/12376-nasa-space-shuttle-program-facts-statistics.html.

[24] SPACEDAILY. More than 1200 satellites to be launched over the next 10 years, October 2013. www.spacedaily.com/reports/More_Than_1200_Satellites_To_Be_Launched_Over_The_Next_10_Years_999.html.

[25] TANIA BRANIGAN, I. S. China unveils rival to international, October 2013. www.theguardian.com/world/2011/apr/26/china-space-station-tiangong.

[26] UCSUSA. Usc satellite database, October 2013. www.ucsusa.org/nuclear_weapons_and_global_security/space_weapons/technical_issues/ucs-satellite-database.html.

[27] 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.

[28] VINCENT L. PISACANE, H. K. U. Embedded software systems. In *Fundamentals of Space Systems Second Edition*. Oxford University Press, New York, 2005.

[29] ZAK, A. Russia 'to save its iss modules', October 2013. news.bbc.co.uk/2/hi/science/nature/8064060.stm.