A Software Approach for Protecting Embedded Systems Memory from Single Event Upsets

Jiannan Zhai, Yangyang He, Yuheng Du, Jason O. Hallstrom
School of Computing, Clemson University, Clemson SC, 29634-0974 USA
{jzhai, yyhe, yuhengd, jasonoh}@clemson.edu

ABSTRACT

Radiation from radioactive environments, 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 real-life applications are used to verify our approach, and the time and space overhead is evaluated.

Keywords

Embedded systems, single event upset, memory protection

1. INTRODUCTION

Humans have a longstanding curiosity about outer space. Since the launch of the first artificial satellite, Sputnik 1 [11], in 1957, over 6,000 satellites have been launched into space. There are more than 1,000 operating satellites in orbit around Earth [15] today, and an estimated 1,200 satellites will be launched over the next decade [13]. As of 2012, more than 130 manned spacecraft have been launched by the United States [10]. There are currently two operational space stations, and seven more are planned over the next decade [9] [4] [17] [14]. A total of 355 astronauts from 16 different countries have flown into space, among which 14 were killed during accidents [12]. Because of 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 extremely high.

One of the most important factors that affect the reliability of spacecraft (and other space equipment) is the quality of constituent embedded systems which control telemetry systems, command systems, attitude control systems, and more [16, 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 [16, 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 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 [8].

The environment outside the Earth's atmosphere is highly radioactive. The radiation is mostly generated by the sun and space itself, and can cause damage to semiconductor devices [16, 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 [5]) 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 [16, p.637]), known as the *Single Bit Upset*. It also has been observed that two or more stored bits can be changed by one high-energy particle, known as the *Multiple Bit Upset*. 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 introduce additional hardware to the target system. Triple modular redundancy (TMR) [16, p.645] uses three identical systems to simultaneously perform identical tasks, and the correct result is decided by a majority vote. N-version programming involves two or more teams developing software for identical requirements. When applied in a TMR context, the intuition is that independent teams are not likely to introduce identical bugs [16, p.646]. A hardware watchdog timer can be used to check execution flow to detect abnormal program executions caused by SEUs [16, p.648]. (More will be added after related work is done.)

In this paper, we present an approach that detects and corrects SEUs in RAM. Our approach focuses on the Single

Bit Upset, which constitute about 3% of the memory errors caused by the SEU. The paper focuses on the system stack, which is the most important and dynamic region in memory. Each callee computes and saves the checksum of its caller's current stack frame, and duplicate 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 conceptually language and application neutral. To demonstrate our approach, an AVR microprocessor, ATmega644 [2], is used in the paper.

The main contributions of our work are as follows: (i) We present a finite-state machine that represents the execution sequence of AVR assembly code generated by AVR-GCC [3]. Each state in the state machine represents a phase in the execution sequence, such as initializing the function stack, calling a function, etc. (ii) We present an algorithm used to scan the AVR assembly code, categorize operations performed by each line of the assembly code based on the state machine. Based on scan results, customized assembly code is injected into the target assembly code, handling register saving and restoring, checksum computation, memory duplication, etc. (iii) Finally, we consider three real-life applications to verify our approach, and evaluate its performance using an AVR simulator.

Paper Organization. Section 2 summarizes key elements of related work. Section 3 discuss the background knowledge related to our approach, including the microprocessor architecture, AVR-GCC, etc. Section **??** presents the design and implementation of our approach. Section **??** presents an evaluation of the approach using three applications. Finally, Section **??** concludes with a summary of contributions and pointers to the future work.

2. RELATED WORK(YUHENG)

3. BACKGROUND(YANG)

Our approach is based on Atmel AVR Toolchain and focuses on AVR microprocessors. In this section, we discuss the necessary background knowledge related to our approach, such as AVR architecture, AVR Toolchain, etc.

3.1 AVR Architecture

AVR microprocessors are based on the Modified Harvard architecture [1], which stores instructions and data in physically separate memories, flash memory and SRAM, and concurrently accesses instructions and data through separate memory buses. The flash memory, which is non-volatile and with high capacity but slow access speed, is used to store executable programs composed of AVR instructions. The SRAM, which is volatile and with low capacity but fast access speed, is used to store data used by the executable programs. For example, the ATmega644 used in this paper consists of a 64KB flash memory, a 4KB SRAM, a 16-bit instruction bus and an 8-bit data bus. Below is a description of the SRAM, as well as the registers, in the AVR microprocessor used in our approach.

```
1  int main(void)
2  {
3     ...
4     foo(arg1, arg2, ..., argM);
5  }
6
7  void foo(arg1, arg2, ...,argM)
8  {
9     int var1, var2, ..., varN;
10    ...
11 }
```

Listing 1: Assembly Code Example

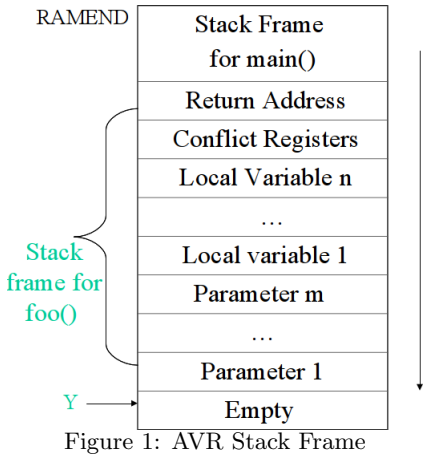


Figure 1: AVR Stack Frame

3.1.1 AVR SRAM

The address range of on-board SRAM of ATmega644 is from 0x0100 to 0x10FF, which can be extended to address 0xFFFF by using external RAM, shown in Figure 3. The AVR SRAM is partitioned into sections, each of which is used to store different types of data. The *.data* section is used to store the initialized static variables and global variables. The *.bss* section is used to store the uninitialized static and global variables. The size of pre-allocated SRAM is the sum of the sizes of .data and .bss sections, which are decided at compile time. The remaining space in SRAM is shared by the heap and stack sections. The *heap* section is used to store dynamically allocated memory when function *malloc()* is called [6]. The *stack* segment is used to store return address, actual parameters, conflict registers and local variables and other information. The *stack* can be used both by the internal control from microprocessor as well as the programmer to store data temporarily. The stack operates with a Last In First Out (LIFO) mechanism, which grows downwards, towards to lower address.

3.1.2 Stack Frame

Inside the stack are stack frames, each of which is a region in the stack used by a function. The stack frame is created when a function is called and is freed when the function returns. For example, as shown in Listing 1, when function *main* calls function *foo*, the space allocation process for function *foo* starts. First, the return address of function *foo* will be pushed into the stack, followed by the conflict registers. Then the local variables, and parameters of function *foo* will be pushed into stack in reversed order. After these

Figure 2: AVR I/O Registers

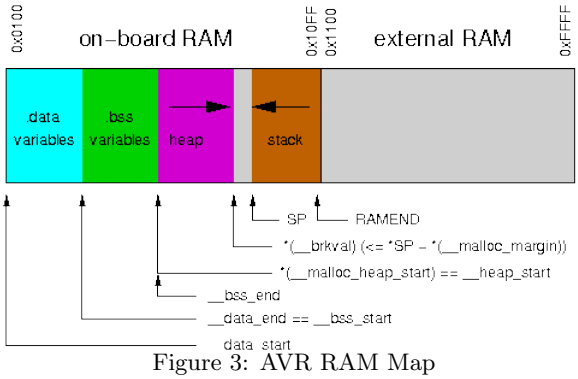


Figure 3: AVR RAM Map

steps, the space that the function call needed has been allocated. The stack frame of function *foo*, from the return address through with first parameter, is created. The stack frame pointer will point to the next available address in the stack as well as the Stack Pointer (SP). When the function *foo* finishes execution and need to return, the stack frame will be freed while the stack frame pointer and the Stack Pointer will point back the next available position after the stack frame of function *main*.

3.1.3 Registers

As shown in Figure x, the General-Purpose Registers, R1 to R32, are mapped into the first 32 bytes of the SRAM, and can be directly used in assembly commands to access and store instructions in the SRAM. Some General-Purpose Registers are used as a pair for special purpose. For example, R29 and R28, the Y-Pointer, is used to indicate the Stack Frame, which will be explained in the next subsection. The operations to the different registers are compiler-specified. For example, the AVR-GCC compiler uses R24 and R25 to store the return value of a function call. Since wrong operations to the registers can bring fatal errors or high cost, it is important to limit the number of registers manipulated by our approach and manipulate them in an appropriate way.

The 64 I/O Registers, written as *0x00* through *0x3F*, are mapped into the next 64 bytes of the SRAM, and actually present the address section from *0x20* to *0x5F* in the SRAM, shown in Figure 2. Some I/O Registers are used as a pair for special purpose. For example, *0x3E* and *0x3D*, are used as the Stack Pointer (SP), which indicates the current top of the stack.

3.2 Function Call Process

Function calls follow the same process, and use the system stack to perform most operations, demonstrated in Figure 4. Figure 4a shows the execution process when a function is called, with each operation labeled with a number. Figure 4b shows the system stack changes after each operation is performed. The numbers below each stack show the operations that change the stack. `SP` is the stack pointer, and `Y` is the stack frame pointer. When a function is called, the return address is automatically pushed into the stack by one of the function call instructions, `call`, `rcall` and `icall` (step 1. 1 is the operation number, similarly hereinafter). After the stack frame pointer is pushed (step 2), the stack frame of the function is created by change the stack pointer and stack frame pointer (step 3). The arguments and local variables are then pushed into the stack (step 4), and the function starts to execute (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` and `reti`, is called.

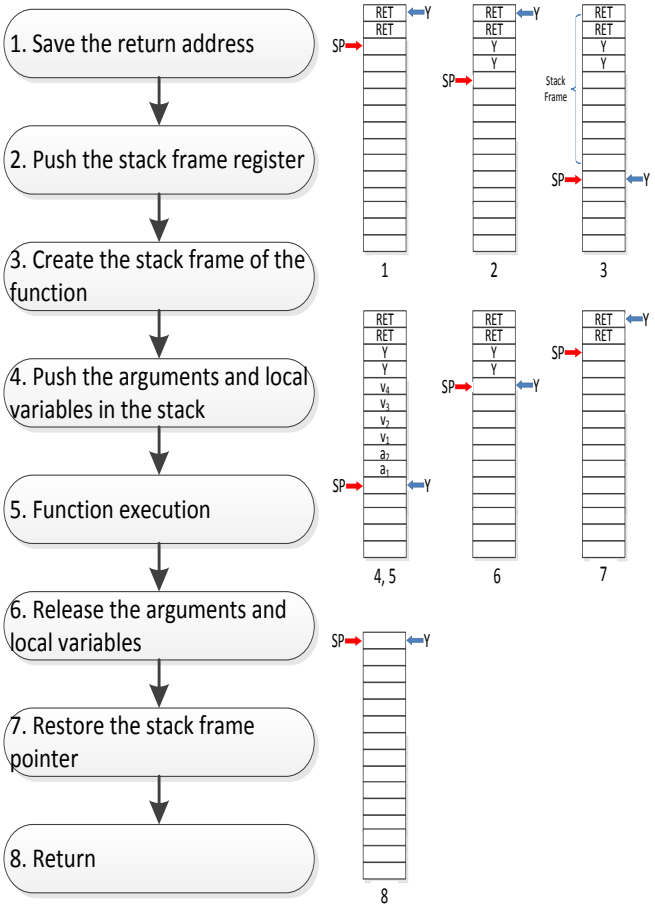
3.3 Atmel AVR Toolchain

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

- *avr-gcc*, an extension of the GNU GCC, is a cross compiler which translates a high-level language, e.g., C and C++, to assembly code for the AVR microprocessors.
- *avr-as* is the assembler which translates the assembly program to object file for the AVR microprocessors.
- *avr-ld* is the linker which uses the Linker Script to combine object modules into an executable image suitable for loading into memory of the AVR microprocessors. By using customized Linker Script, the default memory structure of the AVR microprocessor can be changed and new data structures can be added.
- *avr-libc* is a standard C library which contains many standard C routines as well as many additional AVR-specific library function.

As a matter of convenience, the AVR Toolchain could be used to compile, assemble, and link C program in one command. But to modify the code of the AVR applications in assembly-level and use customized Linker Script, these steps need to be done individually.

When the avr-gcc compiler is used to compile the source code, C program, to assembly program, there are 5 controllable optimization levels [7]. We implement our approach at level **O0**, which is without optimization. since the generated assembly code will show the original behavior of the source program without optimization, the operations to stack can be displayed more obviously. Meanwhile, it would be more convenient for developing, debugging, and evaluating.



(a) Process (b) Stack

Figure 4: Function execution

4. SYSTEM DESIGN/IMPLEMENTATION

Since our approach focuses on the stack in RAM, we make the following assumptions. i) The flash memory and registers are not affected by SEUs. ii) Two or more flipped bits cannot concurrently exist in RAM. iii) The stack frame of the current function is not affected by SEUs. The reason for the assumptions is that in a single-processor system, the detection and correction of a given memory region depend on system registers and other memory regions, such as stack, .data section, etc. If they are affected by SEUs, the detection and correction process cannot work correctly. Moreover, it is rare that two or more bits get flipped at the same time, as mentioned in [find something to support this].

Our approach protects the system stack from being affected by SEUs, by injecting assembly code into the assembly code generated from the target C source code. The code is injected at both the beginning and the end of each function, and handles CRC calculation, memory duplication, etc. When a function is called, the code injected in the beginning of the callee calculates the CRC of the caller's stack frame and saves the CRC and caller's stack frame. Before the callee returns, the code injected in 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 two CRCs do not match.

The `ASM Handler`, a tool written in Java, is created to handle the code inject, shown in Figure **??**. First, the target C source code is compiled to assembly code by GCC. Again, the optimization level is set to `none`. Then, the ASM Handler scans the assembly code and injects customized assembly code into it based on the state machine. Finally, the modified assembly code is assembled and linked into the AVR executable code. In this Section, we discuss the architecture of the ASM Handler, the state machine, and the injected code.

4.1 The ASM Handler

The ASM Handler handles the code inject. It consists of three modules: the `Reader`, the `Scanner`, and the `Injecter`, shown in Figure **??**. A metadata is created to assist categorizing and injecting code. Below is a description of the metadata and each module of the ASM Handler.

4.1.1 ASM Metadata

Each line of assembly code is associated with a metadata, which classifies the code into 3 categories, shown in Listing **??**. A `directive` is used to specify assembly code information, such as system architecture (line 1) and section (line 2), define label (line 3) ,label type (line 4), etc. A `label` is used to identify a location in the assembly code (line 5). In this example, label `main` specifies a location where the main function starts. A `instruction` is used to identify a instruction that will be executed by the microprocessor (line 6-8). The metadata also stores the code inject information, which specifies whether code is injected after this line, and the type of code to be injected.

4.1.2 Reader

The Reader is used to read the assembly code file and generate a metadata list. It reads each line of the assembly code

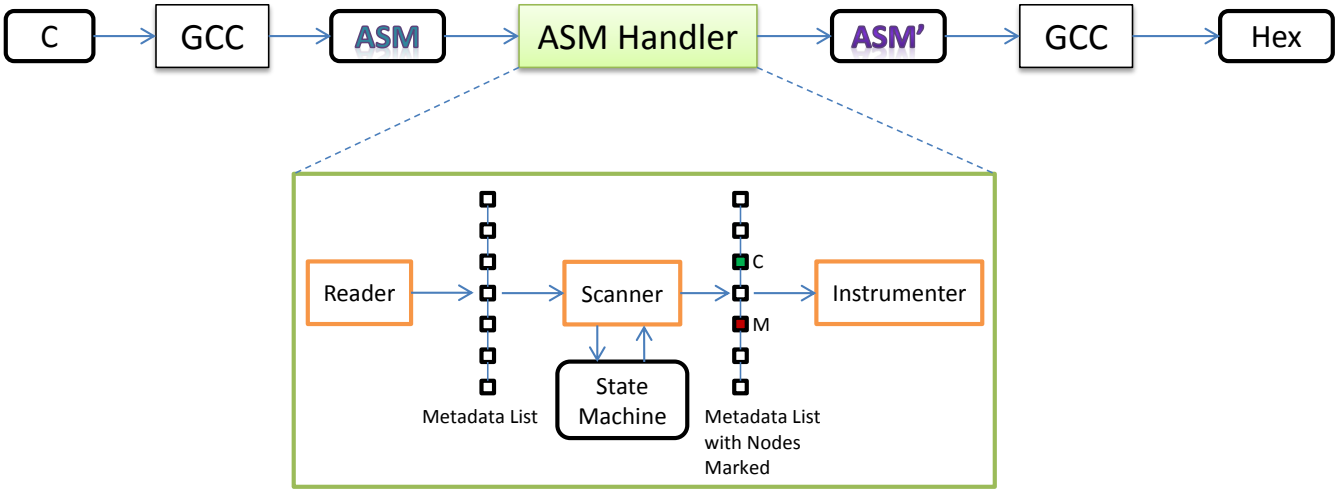


Figure 5: Code injection Process

```
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 2: Assembly Code Example

and generates a metadata node based on the assembly code. The metadata node is then appended to metadata list. For example, the Reader generates a list with 7 nodes after it reads the assembly code in Listing ??, shown in Figure ??.

4.1.3 *Scanner*

The Scanner is used to scan the metadata list. Each metadata node is scanned and passed to the state machine by the Scanner. Based on the current state of the state machine after the metadata node is passed, the scanner either processes the next node, or marks the current node with parameters, indicating if code will be injected before or after the node, and what code will be injected. For example, in Figure ??, two nodes are marked with parameters `C, B` and `M, A`, which respectively indicate CRC calculation and memory duplication code will be injected before and after corresponding nodes.

For each function's assembly code scanned, the Scanner also extracts two parameters. i) `Use_Heap`, which is a boolean, is obtained by scanning if the `malloc` instruction is used in the target code. It indicates whether the `heap` section in RAM is used, determining the RAM section size used to store the stack frame duplicates, discussed in Section ??. ii) `Stack_Frame_Size` is obtained by scanning the assembly code used to establish the stack frame, "`sbiw r28, n`", yielding a stack frame of size $n+6$. The `n` bytes are used to store the arguments and local variables, and the additional 6 bytes are used to store the return address, CRC, and the stack frame size, each of which takes 2 bytes.

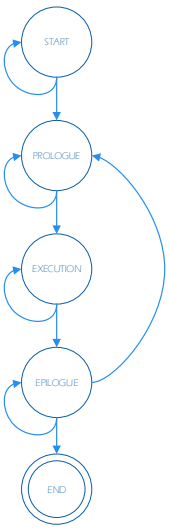


Figure 6: Finite State Machine

4.1.4 *Injecter*

The Injecter is used to inject assembly code into the target assembly code, handling register saving, CRC calculation, memory duplication, etc. It scans the metadata list again, and when a marked node is scanned, the Injecter injects the corresponding code after the line that the node is attached to, based on the parameters stored in the metadata node. Finally, a modified assembly code file is generated, which will be assembled and linked to an executable file.

4.2 Finite State Machine (Yang)

A finite state machine with five states is used to determine where and what customized code segment should be injected into the original assembly program, shown in Figure ??.

4.2.1 *INIT*

INIT is the start state of the finite state machine where the assembly program defines macros, global variables, and performs initialization operations such as xxx. The finite state machine will remain in *INIT* state until the declaration of a new function, for example, ".global foo", is detected. A

code segment which initializes the **SFS** and **STP** sections will be introduced before the state machine shifts its state to *PROLOGUE*.

4.2.2 PROLOGUE

PROLOGUE is the state where a function backs up conflict registers and allocates space for stack frame. In this state, if "sbiw r28,n" is scanned, the result of n added by 6 will be stored as the size of the modified stack frame for later use. The finite state machine will remain in *PROLOGUE* state until the end of the function prologue, "out __SP_L__,r28", is detected. A code segment which pushes the size of he modified stack frame to the stack will be introduced below this line. And if the function being processed currently is not the *main* function, a code segment which calculates the CRC of the caller function, copys the contents in the stack, and updates the **SFS** and **STP** sections will also be introduced before the finite state machine shifts its state to *EXECUTION*.

4.2.3 EXECUTION

EXECUTION is the state where a function finishes its main feature. The finite state machine will remain in *EXECUTION* state until the beginning of the epilogue, "adiw r28,n", is detected. No additional code will be introduced before the finite state machine shifts its state to *EPILOGUE*.

4.2.4 EPILOGUE

EPILOGUE is the state where a function recovers the conflict registers and the stack. A code segment which calculates the CRC, compares the newly-calculated CRC with the CRC stored in stack, recovers the memory and updates the SFS and STP information will be introduced above the line "ret". The finite state machine will remain in *EPILOGUE* state until the last line of the function epilogue, fox example, ".size *foo*, .-*foo*", is detected. If there is no new line afterwards, the finite state machine will shift to the final state. Otherwise, the finite state machine will shift back to *PROLOGUE* state.

4.3 CRC

Input? Output? Discuss the CRC algorithm used.

4.4 Supporting Memory sections

To store the stack frame duplicates, two new sections are created after the `.bss` section by modifying the linker script [**?**], shown in Figure xxx.

The `md` section is used to store the stack frame duplicates, and is design to be a LIFO (Last-In-First-Out) stack structure, called SFS (`Stack Frame Snapshots`). The heap section grows towards the stack, and the usage of the heap and stack during runtime is unpredictable. To avoid the `md` and `heap` sections overlap each other, the size of the `md` section is fixed and is determined by the `Use_Heap` parameter. If `Use_Heap` is `true`, which indicates that the heap is used in the target program, 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 take 1 kilobytes in a RAM of 4 kilobytes, the space available is 3 kilobytes, so the size of the `md` is set to 1 kilobytes.

The `sp` section is used to store the address of the next available memory space of the SFS (similar to the stack pointer), called STP (SnapShot Top Pointer), because a stack pointer is needed for a stack data structure. To protect the STP from being affected by SEUs, the size of the `sp` section is set to 6 bytes and 3 STP duplicates are stored in this section. Because we assume that two or more flipped bits can not exist in RAM at the same time, only one STP duplicate could be altered by the flipped bit. The altered STP is easily excluded by comparing the values of the three STP duplicates, yielding the correct STP value.

4.5 Modified Function Execution Process

Additional operations are added to the original function execution process by the injected code sections, performing CRC calculation, memory duplication, and other supporting operations such as register saving and restoring. Since the code sections are injected into the beginning and end of a function, the modified function execution process can be categorized into two categories, `Modified Function Entering Process` and `Modified Function Return Process`. Below is a description of the modified function execution process.

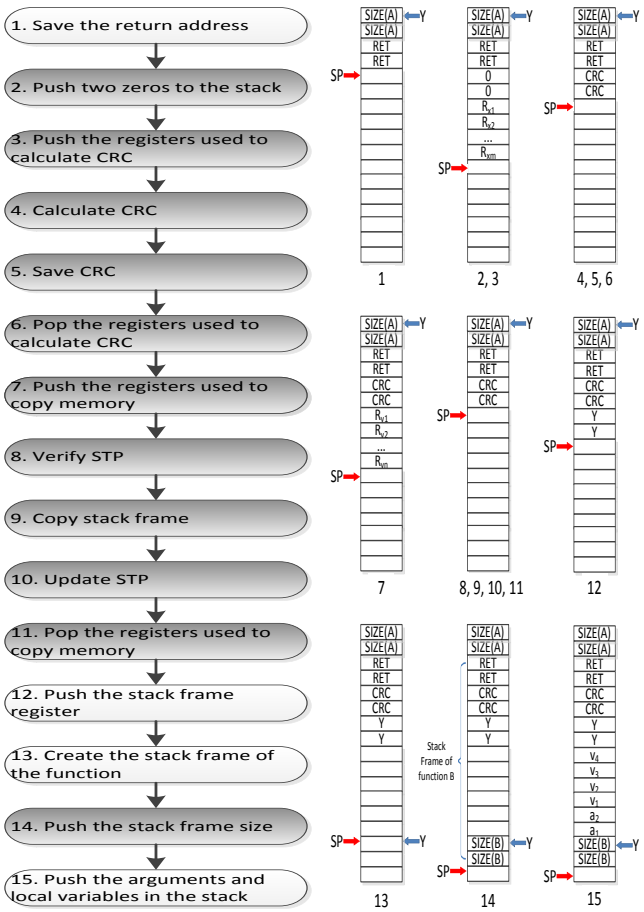
4.5.1 Modified Function Entering Process

The code segments injected into the beginning of a function is used to calculate CRC and save the duplicate of a given memory region, shown in Figure **??**. Figure **??** shows the execution process of the Pre-Execution Code; Figure **??** shows the stack changes based on the Pre-Execution Code. 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. Each operation is labeled with a number. In the stack change diagram, `SP` is the stack pointer, and `Y` is the stack frame pointer. The numbers below each stack show the operations that changed the stack.

When function `B` is called by function `A`, the return address is pushed into the stack automatically by the function call instruction (step 1). To calculate CRC, multiple registers are used, so they must be saved before the CRC calculation process and restored when the process is finished. To avoid the calculated CRC saved in the registers being overwritten when the registers are restored, two bytes (zeros) are pushed into the stack as a placeholder (step 2) for the CRC result before the registers used to calculate CRC is 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 CRC are then restored (step 6).

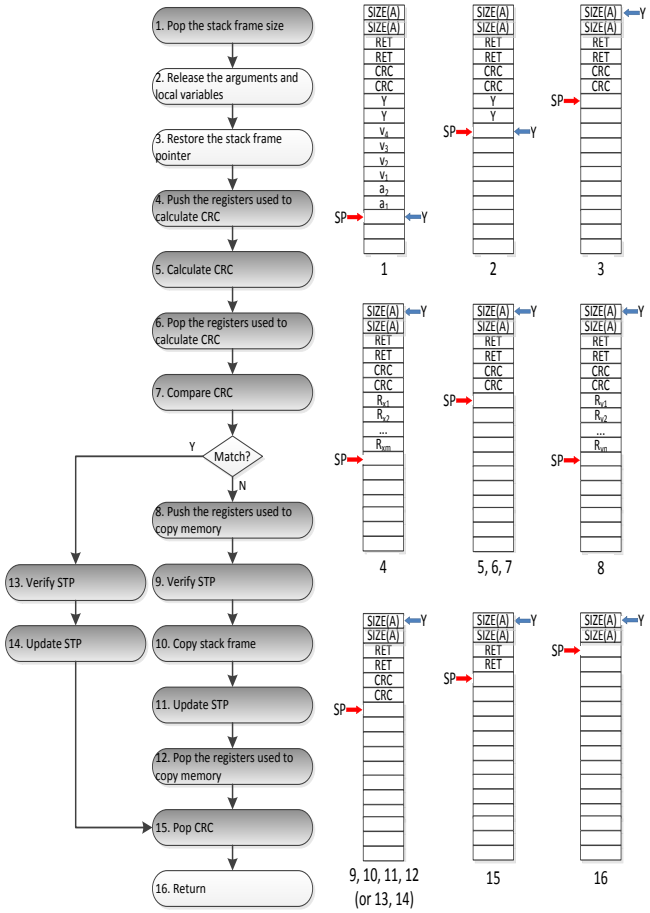
Next, the stack frame of the caller, function `A`, has to be saved. The registers used to save the stack frame are pushed into the stack (step 7). Then, the correct STP is selected by comparing the values of the three STP duplicates (step 8). Using the correct STP, the specified memory is then copied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

After the stack frame pointer of function `B` is saved (step 12) and the stack frame is established (step 13), the stack frame



(a) Process (b) Stack

Figure 7: Function Pre-execution



(a) Process Post (b) Stack Post

Figure 8: Function Post-execution

size of the callee, function `B`, is pushed into the stack (step 15), which is a key operation in the injected code.

When a function is called, the return address is pushed into the stack which is used when the function returns. However, the callee function does not have context information about its caller, including the caller's stack frame address and size. It is thus impossible for the callee to calculate the CRC of the caller's stack frame and duplicate the stack frame without the information of the caller's stack frame size. To solve this problem, each function saves its stack frame size in the stack, which is used by its callee to perform CRC calculation and stack frame duplication.

4.5.2 *Modified Function Return Process*

The code segments injected into the end of a function is used to verify the stack frame of the caller function and restore the stack frame is an SEU is detected, shown in Figure **??**. Figure **??** shows the execution process of the Post-Execution Code; Figure **??** shows the system stack changes based on the Post-Execution Code. 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. Each operation is labeled with a number. In the stack change diagram, `SP` is the stack pointer, and `Y` is the stack frame pointer. The numbers below each stack show the operations that changed the stack.

When function `B` returns, it first pops out its stack frame size from the stack (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 (step 4-6). 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 to the stack and the STP is updated to release the space used to store the stack frame of `A` (step 8-12). Again, the stack frame size of function `A` saved in the stack is used in CRC compare and stack frame restoration. If the two CRCs match, the STP is updated (step 13-14). After the verification of `A`'s stack frame is completed, the CRC is popped out of the stack (step 15). Finally, function `B` returns, and the return address is popped automatically (step 16).

5. EVALUATION

6. CONCLUSION

Future Work External RAM; Memory duplicates compression; multidimensional parity checks.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] P. V. Argade and M. R. Betker. 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] AVRFreaks. Avr gcc/avr gcc toll collection, October 2013. www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC/AVR_GCC_Tool_Collection.

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

[5] P. Darling. Intel to invest more than 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.

[6] S. Goldt, S. van der Meer, S. Burkett, and M. Welsh. 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.

[7] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

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

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

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

[11] A. A. Siddiqi. In *Sputnik and the Soviet Space Challenge*, page 155. University Press of Florida, Gainesville FL, 2003.

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

[13] 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.

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

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

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

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