Microcontroller Compiler-Assisted Software Fault Tolerance

Matthew Bohman, Benjamin James, Michael Wirthlin, Heather Quinn, and Jeffrey Goeders

***Abstract*—Commercial off-the-shelf microcontrollers can be useful for non-critical processing on spaceborne platforms. These microprocessors can be inexpensive and consume small amounts of power. However, the software running on these processors is vulnerable to radiation upsets. In this work, we present a fully automated, configurable, software-based tool to increase the reliability of microprocessors in high radiation environments. This tool consists of a set of open-source LLVM compiler passes to automatically implement software-based mitigation techniques. We duplicate or triplicate computations and insert voting mechanisms into software during the compilation process, allowing for runtime error correction. While the techniques we implement are not novel, previous work has typically been closed-source, processor architecture dependent, not automated, and not tested in real high-radiation environments. In contrast, the compiler passes presented in this work are publicly available, highly customizable, and are platform- and language-independent. We have tested our modified software using both fault injection and through neutron beam radiation on a Texas Instruments MSP430 microcontroller. When tested by a neutron beam, we were able to decrease the cross-section of programs by 17–29x, increasing mean-work-to-failure (MWTF) by 4–7x.**

***Index Terms*—Software fault tolerance, single event upset (SEU), silent data corruption (SDC), soft errors.**

I. Introduction

Commercially produced microprocessors are cheaper, smaller, faster, and more power efficient than radiation-hardened microprocessors. Simple microcontrollers are especially attractive for non-mission-critical processing because of their low cost and power requirements. However, they are vulnerable to radiation-induced single-event upsets (SEU) or soft errors. One of the most insidious forms of an upset is silent data corruption (SDC) where data is altered without warning, causing a cascade of incorrect computations.

There are two general ways to mitigate the effects of soft errors. Hardware-based mitigation aims to modify the architecture or fabrication technology of the processor. On the other hand, software-based mitigation modifies the program to detect or correct errors. Since this work aims to use commercial off-the-shelf (COTS) microcontrollers, we focus entirely on the latter method, software mitigation of faults.

M. Bohman (mbohman@byu.edu), B. James (b_james@byu.edu), M. Wirthlin (wirthlin@byu.edu), and J. Goeders (jgoeders@byu.edu) are with the Dept. of Electrical and Computer Eng., Brigham Young University, 459 CB BYU, Provo, UT 84602 USA, and the NSF Center for Space, High-performance, and Resilient Computing (SHREC).

H. Quinn (hquinn@lanl.gov) is with Los Alamos National Laboratory, ISR-3 Space Data Systems, Los Alamos, NM, 87545 USA

This work was supported by the I/UCRC Program of the National Science Foundation under grant 1738550.

This work was also supported by the Los Alamos Neutron Science Center (LANSCE) which provided beam time under proposal NS-2017-7574-F.

A common form of SEU mitigation is to modify the original software code so that compute operations are duplicated or triplicated. Duplication with compare (DWC) allows for detection when an SEU occurs [1], while triplication with periodic voting allows for automatic correction of single-bit data corruptions [2], [3].

Although these techniques typically slow down program execution, they can be very effective at reducing the error rate, resulting in an increased mean work to failure (MWTF) [4]–[7]. For scenarios where latency is not critical, these program replication techniques can provide much higher reliability for COTS microcontrollers.

Although there is substantial previous work that has utilized these software protection techniques [4]–[20], the work is not easily accessible. Some past work has used hand-modified assembly code rather than an automated process, other works target only specific architectures, assembly languages, or processor features, making them of limited use for future research or commercial projects. Most of these techniques have been tested only in simulation, and not in actual high-radiation environments. In addition, from what we can tell, none of these previous works are currently available as public, open-source tool flows.

In this work we present COAST (COmpiler Assisted Software fault Tolerance), a public tool suite containing several configurable compiler passes that can be used to automatically add runtime protection to a software program. The primary contributions of this work are:

1) An open-source tool flow, built as passes for the LLVM compiler framework [21]. Since the passes operate on LLVM intermediate representation (IR), they are language and target-architecture independent. The tool is publicly available at https://github.com/byuccl/coast. The tool is designed to be easily adopted by other researchers and engineers. In-code directives can be used to control which parts of a program are protected, and several command-line options are provided to control how protection is implemented in the program. Furthermore, we perform validation testing against a wide range of programs to ensure that modified programs remain functionally correct.
2) Experimental testing of this tool on MSP430 microcontroller code, where hardened code was tested both with fault injection and in a neutron beam at the Los Alamos Neutron Science Center (LANSCE). In neutron beam testing, our triplicating protection pass provided a 17–29x reduction in cross-section and 4–7x increase in MWTF.

TABLE I: Rules for Data Flow Techniques [8]

Global Rules	
G1	Each register has a replica
Replication Rules	
D1	All instructions
D2	All instructions, except stores
Checking (Synchronization) Rules	
C1	Before each read on the register (excluding load/store and branch/jump instructions)
C2	After each write on the register
C3	The register that contains the address before loads
C4	The register that contains the datum before stores
C5	The register that contains the address before stores
C6	Before branches or jumps

This paper is organized as follows. Section II provides background information, describing established software protection techniques, previous work, and a description of the LLVM compiler infrastructure. Section III describes our software protection tool, including a brief description of the configuration options, major challenges we faced in automating the protection techniques, as well as how we verify that our tool maintains functional correctness of the programs. Section IV describes our experimental testing to verify the effectiveness of our tool; this includes both fault injection and neutron beam testing results. Section V provides conclusions.

II. Background

A. Code Replication for Software Fault Mitigation

Software-based mitigation techniques exploit temporal and spatial locality to improve radiation tolerance. Variables are stored at separate locations and updated at different times, reducing the probability that a radiation-induced upset will cause SDC. These techniques can be applied to either the data flow or control flow of the program.

EDDI (Error Detection by Duplicated Instructions) [1] is a technique designed to protect the data flow. It performs fine-grained duplication of the program execution, duplicating each data-processing instruction. The two instruction flows are synchronized periodically, checking for data mismatch. Synchronization is performed at least as often as every control flow point or data store, allowing the program to execute with duplicated data, but only a single control flow path. This technique allows the software to detect errors, at a cost of code size and execution time. This technique is also referred to as Duplicate With Compare (DWC).

Chielle et al. [8] enumerate different combinations of program protection. This includes options to indicate which registers will be duplicated, which types of instructions will be duplicated, and at what points in the program the data duplication will be synchronized with the original data. These rules are summarized in Table I. Since the introduction of these rules, many subsequent works have followed the same rules organization [6], [9], [10].

Duplication rules D1 and D2 dictate whether all instructions will be duplicated, or if store operations should be excluded. When store operations are included in the duplication, it effectively results in complete duplication of all variables stored in memory. This is useful when the system lacks memory protection, such as error correcting codes (ECC), which may often be the case with COTS microcontrollers. Chielle et al. [8] determine that the combination of C3, C4, C5, and C6 provide the most reliability for the lowest overhead.

Another form of fault tolerance is triplication of instructions (rather than simply duplicating them), as in SWIFT-R [2]. By triplicating operations, synchronization points can be modified to not only detect errors, but to actually correct errors and continue execution through use of a voting mechanism. This is analogous to triple modular redundancy (TMR) in hardware. This, of course, introduces even larger overheads in terms of program size, memory usage, and execution time. In some cases this overhead can be mitigated; for example, Quinn et al. [5] present a way to execute subroutines a third time only if necessary. However, this optimization technique works exclusively for function-level coarse-grained triplication, and cannot be used for instruction-level duplication as was done in previous works [2], [8].

There are also multiple methods available to monitor the control flow of the program [9], [11]. For instance, control flow checking by software signatures (CFCSS) [12] requires each block to have a signature, which is constantly checked and updated. The effectiveness of control flow methods is debated. Shrivastava et al. [13] claim that control flow checking via signatures introduces too much overhead for the coverage it provides, lowering the effectiveness of the program. However, control flow protection can be effective when paired with the proper data flow protection technique [6]. Our tool that we describe in the next section provides automated passes for both data duplication and triplication; although we also include the option to use CFCSS, we do not focus on it in this paper.

B. Previous Works on Software Protection

There have been many previous works aimed at using software modification techniques to protect against single event upsets. In 2002, Oh et al. [1] presented techniques that modified *GCC* to perform automatic duplication of program instructions, targeted to a superscalar processor. In the same year, Oh et al. also introduced compiler-aided *control-flow* protection techniques, aimed to detect when faults affected the control flow the program. Over the years, several improvements and modifications to these original ideas have been introduced. In 2005, Reis et al. [3], [20] introduced techniques that combined data flow and control flow protections. In 2006, Chang et al. [2] also used the *GCC* compiler to implement several data flow protection techniques, in this case targeting a PowerPC architecture. Vemu et al. [11] and Shrivastava et al. [13] introduced enhancements to the original control-flow checking techniques.

More recently, most research has elected to use LLVM [21], a highly modularized, open-source compiler infrastructure that allows for more straightforward implementation of independent compiler passes. In 2009, Fetzer et al. [14] used LLVM to automatically insert AN-encoding into programs.

AN-encoding is a form of redundancy where information is multiplied by a fixed constant; if at any point the information is not divisible by the constant, an error is reported. In 2012, Khudia et al. [7] used LLVM to automatically profile code, and insert data protection into the critical sections. In 2016, Didehban et al. [15] introduced nZDC (near Zero Data Corruption), a technique that uses LLVM to apply duplicate with compare logic. While the previously mentioned techniques focus on fine-grained protection mechanisms, other techniques have taken a more coarse-grained approach. For example, Reinhardt et al. [18] and Wang et al. [17] introduced compiler techniques to automatically create redundant threads, providing protection against upsets by taking advantage of multicore architectures. Other works, such as Nakka et al. [19], have proposed processor architectural changes that could be made to support new data protection techniques. All of the previously mentioned approaches [1]–[3], [7], [11]–[15], [17]–[20] have focused on a server-like environment, targeting high-performance, superscalar processors. In contrast, this work focuses on techniques for embedded systems, where the software may be used in a high-radiation environment.

While some of these tools could be applied to embedded systems with microcontrollers, many of them rely on more advanced architectural features. For example, most tools make the assumption that memory and caches are protected by a method such as error correcting codes (ECC) [3], [15], only accounting for errors in the CPU. nZDC obtains a higher performance by utilizing pipeline forwarding and load/store queues. Oh et al. [1] also relied on super-scalar processors to reduce the overhead of fault tolerance. These advanced microarchitectural features are not available to all microcontrollers, so in this work we have focused on basic duplication and triplication of instructions to correct errors without relying on specific hardware mechanisms.

There is another body of work that has focused more on simple duplication and triplication techniques for embedded systems, and evaluating them in high upset environments. In 2012, Chielle et al. [16] introduced the CFT-tool, an automated tool designed for fine-grained assembly-level replication, targeting MIPS and ARM assembly. The user provides an assembly file and information about the target architecture. The CFT-tool then applies the specified mitigation techniques and generates a hardened version of the assembly file. However, directly modifying the assembly, instead of using a compiler-based technique such as the works mentioned above, means that only certain assembly formats and architectures will be supported. Using the CFT-tool, Chielle et al. have produced several works [6], [8]–[10] that have evaluated the different levels of protection enumerated in Table I. Another tool focused on embedded applications is the Trikaya tool, introduced by Quinn et al. [4], [5]. This is also an automated SEU mitigation tool that uses LLVM; however, the initial iterations of the tool operated at a coarse-grained level, only replicated functional calls instead of individual operations, and was never fully completed or released. Preliminary results did not make use of the automated tool, and instead used hand-modified assembly to test the benefits of duplication and triplication.

Of the above described tools [1]–[20], none of the works provided open-source, publicly available tools. Furthermore, only four of the papers ([4]–[6], [10]) present results tested in an actual high-radiation environment, the rest only have only simulated upsets with fault injection.

To the best of our knowledge, this work, while not introducing novel theoretical techniques, provides the first open-source, publicly available software protection tool, that is furthermore tested and validated in an actual high radiation environment. In addition, this work is geared toward public release and use, with validation on a wide set of software programs – not just our benchmark programs.

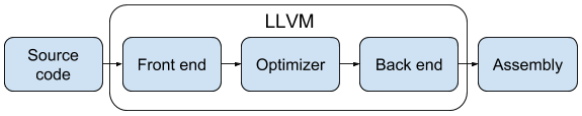


Fig. 1: LLVM Compiler Diagram

C. LLVM

The LLVM compiler infrastructure is an extremely flexible tool for code generation. The tool flow consists of three phases as illustrated in Figure 1.

Front end The front end takes user source code and transforms it to a language- and machine-independent intermediate representation (IR). Several front ends are provided for different languages (C, C++, Swift, Objective-C, etc.).

Optimizer This stage refers to all compiler passes that take IR as input and produce optimized IR as output. This includes, for example, the `-O3` passes.

Back end The optimized IR is used to generate architecture-specific assembly or binary files, targeting for example, x86, ARM, MIPS, MSP430, etc.

LLVM has a number of desirable traits, the foremost of which is that of the three-stage organization. The standard IR language means that we can write one optimization pass and it can be used on a variety of languages and architectures. Other past work focused on a single architecture [3], [17]. Additionally, the IR is human-readable, allowing manual inspection of the replicated code.

III. Implemented Techniques

This section describes COAST, our software protection tool which uses LLVM compiler passes to automatically add data flow protection to user provided programs.

In [16], the authors find that the VAR3 protection scheme (G1, D1, C3, C4, C5, and C6 from Table I), provides the greatest fault coverage for the lowest overhead when targeting COTS microcontrollers. This consists of replicating all registers, all data-processing operations, and all memory loads/stores, while leaving a single set of control flow operations. Synchronization points are inserted at each control flow point or data store. This approach replicates the program data flow while keeping a single control flow. We follow this approach for the default configuration of the tool, although we describe

in Section III-A how our tool can be configured to select different protection rules.

Our protection passes operate on the LLVM IR, automatically duplicating or triplicating instructions and inserting synchronization points. This is done after all other compiler optimizations to ensure that the redundant code is not optimized out by other compiler passes.

The pass begins by first finding all functions that should be protected. It then iterates through every IR instruction in these functions, detecting what instructions to clone and adding them to a list. Once this list is complete then COAST replicates every instruction in the list. Additionally, dependencies between instructions are updated so each clone operates on its own copy of the data. The function signatures are also modified to include the clones of the original arguments. When all of the clones have been inserted into the code, the pass sweeps through the program and detects where synchronization logic should be placed. It then inserts comparison statements and error handlers for DWC, or voter code for TMR.

```
do:
  ld r0 = i


  r1 = sub r0, 1


  r2 = cmp r1, 0




  br neq r2 do
```

(a) Original Code

```
do:
  ld r0 = i
  ld r10 = i_copy
  ld r20 = i_copy2
  r1 = sub r0, 1
  r11 = sub r10, 1
  r21 = sub r20, 1
  r2 = cmp r1, 0
  r12 = cmp r11, 0
  r22 = cmp r21, 0
  r3 = cmp r2, r12
  r4 = select r3, r2, r22
  br neq r4 do
```

(b) TMR code

Fig. 2: Code before and after mitigation pass

An example of the LLVM IR instructions before and after triplication is provided in Figure 2, with unmitigated code on the left, and our mitigated code on the right. The bold text indicates changes made by our pass. In this example the code fetches a value from memory into a register, subtracts 1, and then performs a branch based on whether the new value is 0. In the TMR version, we fetch the three copies of the value from memory, perform a subtraction on each register, then, because the branch is a necessary synchronization point, we compare the three values, vote on the correct value, and use that value in the branch operation.

If the user chooses to use DWC instead of TMR, similar code would be generated, except with only one copy of each instruction instead of two. At the synchronization point there is no voting code, but rather the two data copies are compared, and a user-selectable function is called (`abort` by default) if there is a discrepancy.

A. Configuration Options

A primary goal of our tool was to make it accessible to a wide audience of engineers and researchers. Since different applications have varying needs for protection mechanisms, a central design principle of our tool is that the user has high control over the protection passes through the use of several options. Table II lists some of the available command-line options; a full listing is available in the user manual.

1) Replication Scope: The most important configuration is allowing the user to select which parts of the program should be replicated. The sphere of replication (SOR) [18] is a concept detailing what variables and instructions should and should not be replicated. Although including the entire program in the SOR should maximize the fault coverage, the overhead can be prohibitively high. As has been shown in [19] it is possible to exclude instructions from the SOR without affecting fault coverage. To that end, our tool allows users to explicitly include or exclude functions and variables from the SOR using either the command line or in-code user directives.

The user can specify any functions and global variables that should not be protected using `-ignoreFn` and `-ignoreGlbl`. At minimum, these options should be used to ensure that neither interrupt service routines nor any code that interacts with hardware devices (GPIO, UART) are replicated. Replicating such code is likely to lead to errors. It should be noted that by not replicating I/O code, a user will likely introduce single points of failure into the program. While it would be possible to introduce some mechanisms to tolerate such upsets, such as repeatedly writing to a register and reading back to verify the value, the approach would be dependent on the individual hardware architecture, and so would be up to the user to implement in their code. Since I/O code may lead to single points of failure, users should seek to minimize this code where possible.

In addition to the command line options, our tool allows users to embed directive directly in their source code. When a function, global variable, or local variable is marked in-line with either `__NO_TMR` or `__TMR` it is treated as if it were passed in through the command line. Figure 3 provides an example of user code marked with these directives.

There are a number of challenges we encounter when only parts of the program are selected for replication; these are briefly described later in Section III-B.

```
#include "coast.h"

__DEFAULT_NO_TMR

int matrix1[s][s];
int matrix2[s][s];
int __TMR result[s][s];
int __NO_TMR golden[s][s];

...

void __TMR matrixMultiply (...) {
  ...
}

void __NO_TMR checkGolden (...) {
  ...
}
```

Fig. 3: Example user code employing in-code data protection directives

TABLE II: Selected COAST Command Line Configuration Options

Command line option	Effect
`-noMemReplication`	Don't replicate variables in memory (ie. use rule D2 instead of D1)
`-noLoadSync`	Don't synchronize on data loads (C3)
`-noStoreDataSync`	Don't synchronize the datum on data stores (C4)
`-noStoreAddrSync`	Don't synchronize address on data stores (C5)
`-ignoreFn <X>`	<X> is a comma separated list of the functions that should not be replicated
`-ignoreGlbl <X>`	<X> is a comma separated list of the global variables that should not be replicated
`-DWC`	Use DWC instead of TMR

2) Duplication and Synchronization Rules: While we chose to implement the options D1, C3, C4, C5, and C6 by default (Table I), the first three configuration options listed in Table II allow the user to override these defaults. This controls whether variables in memory are duplicated and when synchronization points are inserted. For example, if the user's system provided ECC memory, it is likely that the user would want to add the flag `-noMemReplication`. In the example code in Figure 3, this would mean that `r0`, `r10`, and `r20` would all be initialized from loading the same memory location, `i`.

Likewise, the frequency of synchronization between the data replicas can be reduced by disabling rules C3, C4, or C5. This would reduce the impact to program performance; however, it may come at a cost of lower fault coverage, or longer latencies to error detection.

C1 and C2 are not implemented by our pass as these were shown in previous work to be an excessive amount of synchronization [8]. G1 and C6 cannot be disabled as these are necessary options for the protection to function correctly.

3) Other Configuration Options: COAST offers several other options, that we mention briefly here. These include:

Error counting The TMR voting logic is modified such that a global variable is maintained that counts the number of times that the voter corrects an error. This introduces greater overhead as the voter must now load and store to the global variable at each synchronization point (although we could update the variable only when an error is detected, this would introduce control-flow branching, and would incur an even greater overhead). Despite this cost, it may still be desirable to collect this data, as we chose to do so during our neutron radiation testing.

Input initialization The user can select to initialize replicated input variables at compile time, or at runtime via automatically inserted calls to `memcpy`.

Library calls For calls to unprotected library code, the user can select whether the call should be replicated (as would be appropriate for `malloc`), or whether a single call should be made and the returned value be passed to all replicas (as would be appropriate for `rand`).

CFCSS The user can enable the CFCSS control flow protection pass. This technique is described in [12].

B. Challenges with Automated Replication

Although many of the previous works have reported automated tools for program protection, these have typically been internally used tools without any public release. As such, they were likely designed to operate on a fixed program style or structure. Since we have designed our tool for public release and use, we have attempted to support arbitrary C code. In this process we have encountered a number of hurdles, such as issues in passing replicated data between functions, crossing the boundary between replicated and unreplicated code, calls to system libraries, and so on. We have not come across any previous work that has described how these cases are handled; we suspect in many cases researchers have formatted their benchmarks to avoid such complications.

Function calls crossing the SOR boundaries cause a series of problems. Figure 4 will be used to illustrate them. This figure shows a series of functions. Functions A and B both call function C in this example.

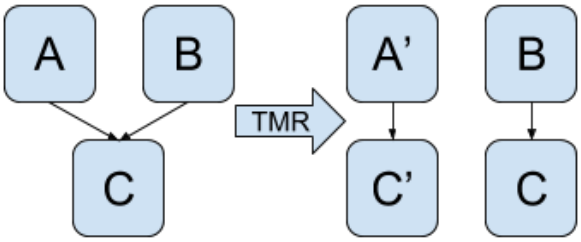


Fig. 4: Problems With Functions Crossing SOR Boundaries

One challenge with code replication is how to treat function calls. For example, suppose function C is called by function A and B. One might assume we could just replicate the call to function C; however, this will end up replicating control flow, whereas we only want to replicate individual operations and data. In addition, this could cause incorrect execution if function C causes side effects. Instead, we modify the signature of function C by replicating each argument. The caller code passes in its data copies to the different arguments. Inside of the function, the replicated versions can then be used. This maintains the original control flow while still providing data replication. This has the downside of passing arguments into the function via the stack if not enough argument registers are available.

Another problem arises when functions are excluded from the sphere of replication. In some cases, a function might be simultaneously included and excluded from the sphere of replication. In Figure 4 suppose functions A and C have TMR applied, but function B does not. When function C is protected, the call signature is modified to add the triplicated arguments. This would invalidate the call to function C that unprotected function B makes. To solve this conflict, we create a copy of function C with TMR applied, denoted C'. The hardened version of function A, A', then calls C'. Function B is then free to call the original version of C. Although this has the negative effect of increasing code size, it would be up to the designer to decide the final configuration, taking into account program size, performance, and fault coverage.

Another problem arises when functions are excluded from the sphere of replication. In some cases, a function might be simultaneously included and excluded from the sphere of replication. In Figure 4 suppose functions A and C have TMR applied, but function B does not. When function C is protected, the call signature is modified to add the triplicated arguments. This would invalidate the call to function C that unprotected function B makes. To solve this conflict, we create a copy of function C with TMR applied, denoted C'. The hardened version of function A, A', then calls C'. Function B is then free to call the original version of C. Although this has the negative effect of increasing code size, it would be up to the designer to decide the final configuration, taking into account program size, performance, and fault coverage.

One of the main points of vulnerability in code protected by our tool is that our pass cannot currently protect library calls because the libraries are precompiled. In order to have protected libraries available, one would have to recompile the libraries from the source code. The default behavior is to replicate all calls to library functions except a few which cause incorrect execution, such as `printf()` or `rand()`. The user has an option to only perform the library call once instead, then use the return value for the instruction clones. This can be controlled via the configuration arguments.

Another limitation of the tool is that return values are not replicated. Instead, a single value is returned by a function and then replicated by the caller function. This, of course, creates a single point of failure. In future work we plan to add a configuration option to automatically modify function signatures to include variables, passed-by-reference in arguments, that could serve as multiple replicated return values.

C. Verifying Correctness

In order to prepare this tool for public release, we wanted to ensure that 1) the tool supported a wide range of software constructs and primitives, and 2) that the modified code produced by our tool remained functionally correct. To ensure this, we test our repository nightly against a suite of self-verifying C code benchmarks. Our benchmark suite contains the following programs: matrix multiply, quicksort, CRC, AES, FFT (4 variants), llvm-stress, MiBench (6 programs), CHStone (12 programs), and CoreMark (2 programs). We also had a few custom unit tests designed to exercise very particular use cases of the protection algorithms. Together these give us over 30 benchmarks to test against, providing a good spread of algorithm types and code sizes.

There are some particular code constructs that the tool does not yet support. It does support single-level pointers (including function pointers); however, multiple levels of indirection are not supported (double pointers). This is because of issues where the synchronization code expects variables to have the same values; however, pointers to different replicas of the data will naturally have different values. Although we can detect this case and handle it for direct pointers, multiple levels of pointers are more challenging and not yet supported. In addition, there are specific C library calls, such as `fscanf()`, that cannot be protected. This is due to some complexities surrounding the way the C library functions handle file I/O.

IV. Experimental Results

This section describes our experimental results. These were obtained by performing fault injection testing on a COTS microcontroller, as well as testing the same chip in a neutron beam at Los Alamos Neutron Science Center (LANSCE). The objective of these tests was to verify that our automated protection techniques are effective and achieve comparable results to past work.

A. Methodology

For our testing (both fault injection and neutron beam) we targeted the FeRAM-based MSP430FR5969 microcontroller [22], on the EXP430FR5969 development board. This part contains 16KB of FeRAM, which contains the program executable, and 2KB of SRAM, which contains the program data. This part was chosen because it has previously been tested in a radiation environment [5] and because the MSP430 family is supported by an LLVM back end, meaning we can compile LLVM IR code into MSP430 assembly code.

For both the fault injection and neutron beam testing we utilized self-checking benchmarks. Each benchmark returned a boolean indicating whether a checksum of the result matched a known golden checksum. There were three possible results.

1) Correct checksum, indicating that execution occurred without data corruption.
2) Incorrect checksum, showing that a fault was activated and an SDC occurred.
3) The program could hang.

We used the following metrics to quantitatively evaluate the increase in reliability and cost of replication. All results are normalized against the unmitigated version of the benchmark.

Code size Change in executable size.

RAM size The increase in the RAM usage. Our benchmarks do not use dynamic memory allocation, so this is a fixed amount allocated at compile time.

Runtime Execution time of benchmark.

Mean Work to Failure (MWTF) [20] The longer runtime of protected programs results in them having a greater chance of encountering a fault during execution. The MWTF metric captures the relationship between reliability and performance in Equation 1.

$$\begin{aligned} \text{MWTF} &= \frac{\text{amount of work completed}}{\text{number of errors encountered}} \\ &= (\text{raw error rate} \cdot \text{AVF} \cdot \text{execution time})^{-1} \end{aligned} \tag{1}$$

Fault coverage The percentage of faults that are detected and corrected. It is calculated using Equation 2:

$$F_{coverage} = \frac{F_{detected} + F_{masked}}{F_{total}} = 1 - \frac{F_{undetected}}{F_{total}} \tag{2}$$

B. Fault Injection

Fault injection is used to characterize the effects of upsets on microprocessors without costly radiation testing [23]. It can be done in simulation [24] or on hardware [6]. It is, however, more limited in scope than radiation testing because faults can

only be injected where the tools allow. The radiation beam has no such limitations.

The goal of our fault injection testing was to automatically inject bit flips into programs executing on our DUT, and measure how often a fault resulted in a hang, SDC, or had no effect. We utilized Texas Instruments' Debug Server Scripting interface [25], which provides an API into the debugging tools for the MSP430. This allowed us to create an automatic fault injection tool where we could execute a program on the hardware, pause it at a random point in time, flip a bit of memory or in the register file, and then continue execution. For this test we targeted the SRAM exclusively.

In order to inject bits at random points in execution it was necessary to pause the program with a high degree of precision. We found that issuing a pause command to the TI Debug Server was too slow. Even if we issued a pause command immediately after starting the program, most of the program would have already executed before it halted. Instead, we utilized a built-in hardware timer and triggered an interrupt upon the timer expiring. By placing a breakpoint on the interrupt service routine, and by configuring the timer period register to random durations throughout the program execution, we could halt the program and inject a fault at random times throughout the entire program execution.

We tested three benchmarks, both with and without automated data flow protection. We used a matrix multiply benchmark (MxM), a cyclic redundancy check (CRC) benchmark, and a quicksort (QS) benchmark. These benchmarks were chosen to be comparable to previous work [4]. The tests were completed with and without TMR mitigation using the default configuration of COAST. Table III contains the results observed after 5000 fault injection runs for each TMR design, normalized to the results obtained without TMR. Each run consisted of programming the executable into the DUT, executing a random duration of time, flipping a random bit of SRAM memory, and running the program to completion.

TABLE III: Fault Injection Testing with TMR Mitigation

Benchmark	Code Size	RAM size	Runtime	MWTF	Fault Coverage
MxM	2.1x	3.0x	2.3x	28.0x	99.3%
CRC	2.1x	3.0x	2.7x	11.4x	98.4%
QS	2.2x	2.8x	3.6x	16.1x	99.0%

Table III list the results of the fault injection testing. The results show a MWTF increase ranging from 11.4–28.0x. As expected this comes at the price of 2.1–2.2x larger code size, 2.8–3.0x more RAM usage, and 2.3–3.6x longer runtime. There is more extensive fault injection data available in [26]; however, it is not included in this article due to space considerations.

At this point in time we have not yet explored specific fault injection cases and why they escape detection using the DWC/TMR techniques. However, we anticipate this will be focused on in future work to motivate future features added to COAST.

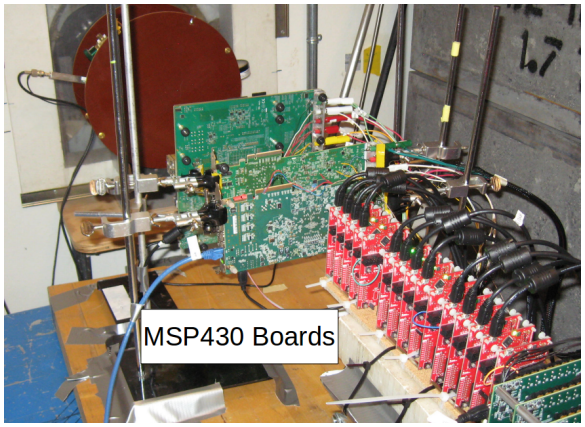


Fig. 5: Picture of LANSCE neutron test. The MSP430 boards are the red boards in the foreground.

C. Radiation Testing

We also tested the mitigation pass in a neutron beam at LANSCE. The purpose of the radiation test was to validate both the operation of the pass and understand how well fault injection predicted real-world behavior in space. Each program was run in the beam repeatedly and the results of each run were logged. The MWTF was determined by dividing the total number of correct executions by the total number of incorrect executions.

The experiment was conducted using 15 MSP430FR5969 boards in the 30L flight path at LANSCE, as shown in Figure 5. Several boards were utilized since the cross-section of each individual MSP430 is quite low, and utilizing multiple boards allowed us to induce a sufficient number of faults to gather statistically significant data. The 15 boards were spread over a distance of $24\,\mathrm{cm}$, with the closest board located $83\,\mathrm{cm}$ from the detector, and the detector located $19.67\,\mathrm{m}$ from the tungsten core. Since the flux reduction follows the ratio $r^2/(r+d)^2$, where r is the distance to the detector, and d is the distance from the detector to the DUT, the closest board in the beam experiences a 7.9% attenuation in flux versus the measurements taken at the detector, whereas the furthest board from the cap experiences a 10.1% attenuation. This 2.2% range in flux between the 15 boards is accounted for in the results, when the data from all boards are aggregated.

Each set of five MSP430 boards were connected to a Raspberry Pi 3 board, which provided control and monitoring. The Raspberry Pi board would program each DUT via USB, after which the DUT would repeatedly execute the chosen benchmark in an infinite loop. At the end of each execution iteration the DUT used GPIO pins to return a status to the Raspberry Pi indicating whether the benchmark executed successfully, whether an SDC occurred (through checking against the golden checksum), and whether TMR had corrected any faults. The Raspberry Pi and DUT would then perform a handshake via GPIO pins to indicate that the Raspberry Pi had logged the status and that the DUT could continue with the next iteration. If the DUT reported an error status, or if it stopped responding, the Raspberry Pi would power cycle the board, reprogram it, and continue the testing process.

TABLE IV: Results from neutron beam test

Benchmark	Total Executions	Fluence (n/cm^2)	Corrected Fault	SDC Errors	Hangs	Code Size (bytes)	RAM Size (bytes)	Runtime (ms)	MWTF	Cross Section (cm^2)
MxM Unmit.	7665806	3.8×10^{11}	N/A	40	5	1752	524	100.4	191644	1.067×10^{-10}
MxM TMR	5410937	1.1×10^{12}	291	4	7	5716	1830	402.3	1352658	3.715×10^{-12}
Change						**↑3.3x**	**↑3.5x**	**↑3.7x**	**↑7.1x**	**↓28.7x**
CRC Unmit.	5924316	2.1×10^{11}	N/A	30	0	1752	568	43.7	197476	1.427×10^{-10}
CRC TMR	3380543	4.8×10^{11}	159	4	3	4288	1972	178.3	845094	8.185×10^{-12}
Change						**↑2.4x**	**↑3.5x**	**↑4.1x**	**↑4.3x**	**↓17.4x**

We tested the matrix multiply and CRC benchmarks in the neutron beam. Each was tested with and without TMR mitigation, again using our default tool configuration. Additionally, the designs with TMR applied also had logic inserted which reported if TMR corrected any faults, which accounts for the additional overhead when compared to Table III. The results are provided in Table IV, and the cross section is illustrated in Figure 6.

As shown in Figure 5, the 15 test boards were located downstream in the beam from three other circuit boards from an unrelated experiment. It is possible that these three boards, and the multiple boards used in the test, which were spaced close together, introduced secondary interactions from the neutron beam. However, our past experiences in such a test environment suggest this effect would be very minimal, and not significant given the moderate error bounds of the data obtained. Figure 7 provides the cross section and fault distribution per board for TMR-detected faults in the matrix multiplication benchmark (this configuration was chosen as it contains the most detected events). As illustrated in the figure, there is no clear indication that the cross-section was dependent on the board position in the experiment.

Triplicating the program operations leads to a 17.4x reduction in error cross section for the CRC benchmark, and a 28.7x reduction for the matrix multiplication benchmark. However, this reduction comes at a performance cost, introducing a 3.7–4.1x slowdown in program execution. The MWTF metric accounts for this slowdown, and results in a 4.3–7.1x increase in mean work that can be accomplished between failures. This means that if the designer is willing to accept a higher program latency, as well as a ˜3x increase in code size and RAM size, the TMR technique can offer much greater levels of reliability.

D. Results Analysis

Overall, our automated TMR tool significantly increased the MWTF in all of our benchmarks. The MWTF improvements are noticeably lower for the radiation test compared to fault injection. This is likely for two reasons. First, the runtime penalty for TMR was greater in our radiation test data, as we enabled the option that allowed for counting the number of faults corrected by the TMR voter (see Section III-A3). This was necessary in order to collect the TMR fault data in Figure 6. While a better comparison to the fault-injection could be made if the fault-injection were configured with the same fault-correction counter enabled, we had not yet developed this feature at the time of our extensive fault injection testing.

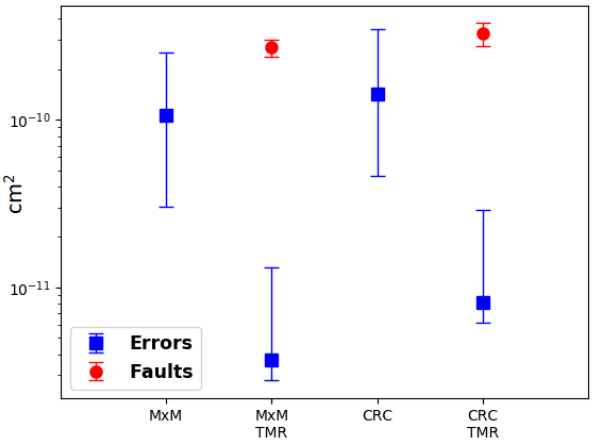
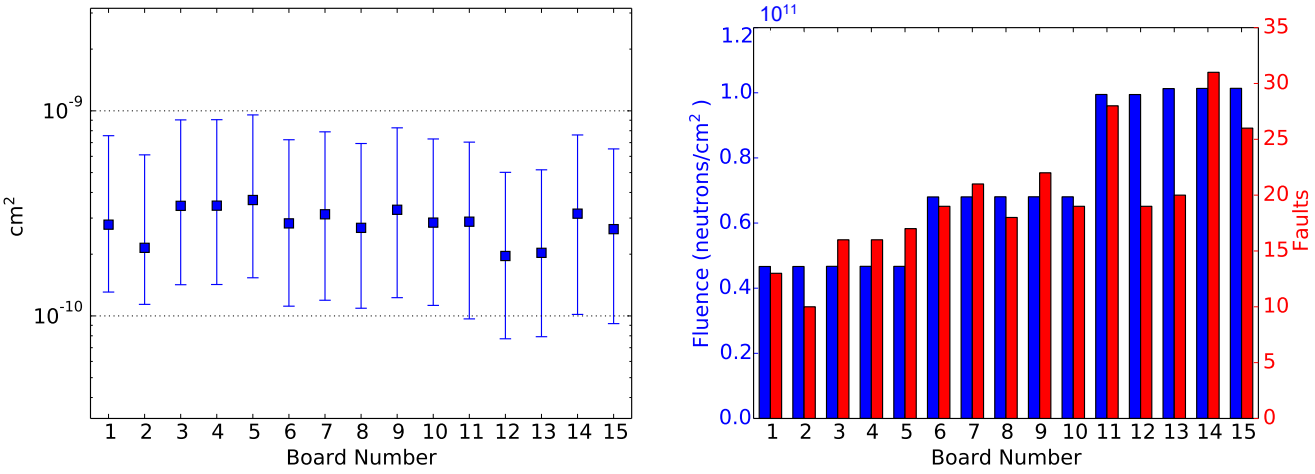


Fig. 6: Design cross sections with 95% confidence interval.

Future experiments will collect further data to make better comparisons between fault injection and radiation test results. Secondly, our fault injection software limits the processor features that can be accessed, whereas the radiation beam has no such limitations.

Compared to work using the CFT-tool [16], we see similar results. COAST's default set of checking rules is reported to have 95% fault coverage, whereas in our fault injection experiments we observed an average of 98.9% fault coverage [8]. Additionally, the same set of rules was combined with a control flow technique and tested in a heavy ion beam, which reportedly gave a MWTF increase of 1.66x [6]. When tested with a neutron beam, we observed an average increase of 5.7x in MWTF. This is just a rough comparison, as there are considerable differences in methodology – for example, the mentioned work does not duplicate variables in memory and instead assumes ECC. Another past work [4] showed that hand coded coarse-grained TMR improved cross section by approximately 8x on a matrix multiplication. Although our methodology is fairly different from these past works, we believe the results show that our automated protection is at least as effective as previous work.

We observed that most of the program hangs during fault injection occurred when the program stack was corrupted. COAST adequately protects the data contained in the SRAM, but it cannot protect copies of the system registers, such as the program counter or stack pointer. When these values are upset the program goes into an indeterminate state. Control flow mitigation (which our tool provides, but we did not have time



(a) Cross sections with 95% confidence interval

(b) Fluence and faults

Fig. 7: Fault distribution per board for the TMR-detected faults, with the TMR configuration of the matrix multiplication benchmark. Board 1 represents the board closest to the beam source (83 cm from detector), while board 15 is the furthest from the beam source (107 cm from detector). Note that the fluence received by each board varied, as some boards were used for other benchmarks or configurations at different times of the experiment.

to test in the beam) would help protect against incorrect jumps to other places in the program. However, very little of the MSP430FR5969's potential address space is occupied by the actual program, so control flow protection is of questionable benefit. Instead, it would be more helpful to have a watchdog timer monitoring the current program counter. If the processor jumped to an invalid address then the processor would reset instead of halting entirely.

V. Conclusion

Commercial off-the-shelf microcontrollers are attractive for non-mission-critical processing in high radiation environments. However, they are vulnerable to silent data corruption. The COAST tool presented in this paper takes existing software protection techniques and applies them to microcontroller software in a fully automated manner. This allows users to quickly apply fault tolerant techniques to their software code, with a high degree of control. We are working towards a public release of the tool. Test results done with both fault injection and neutron beam testing show that COAST provides coverage comparable to the state of the art, increasing the mean work to failure by as much as seven times and decreasing the cross section by as much as 28 times.

TMR provides a high increase in mean work to failure, but the runtime and memory overhead may be prohibitively high. This work began an exploration into automated data flow protection using replication. Future work can delve deeper into other protection methods. One possibility is identifying sensitive parts of the program and applying replication selectively to these parts. We have made provision for the user to do this with command line options and in-code directives, but automatically identifying these sensitive regions may be more desirable.

References

[1] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[2] J. Chang, G. Reis, and D. August, "Automatic Instruction-Level Software-Only Recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 83–92.

[3] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 2005 International Symposium on Code Generation and Optimization, CGO 2005*, vol. 2005. IEEE, 2005, pp. 243–254.

[4] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software Resilience and the Effectiveness of Software Mitigation in Microcontrollers," in *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, Dec. 2015, pp. 2532–2538.

[5] ——, "Robust Duplication With Comparison Methods in Microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 338–345, Jan. 2017.

[6] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt, "Reliability on ARM Processors Against Soft Errors Through SIHFT Techniques," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2208–2216, 2016.

[7] D. S. Khudia, G. Wright, S. Mahlke, D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 99–108, 2012.

[8] E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi, "Overhead reduction in data-flow software-based fault tolerance techniques," in *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*. Cham: Springer International Publishing, 2015, pp. 279–291.

[9] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, F. L. Kastensmidt, R. Reis, and S. Cuenca-Asensi, "Reliability on ARM processors against soft errors by a purely software approach," in *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*, vol. 2015-Decem. IEEE, Sep. 2015, pp. 443–447.

[10] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective Software-Only Error-Detection Technique Using Assertions," in *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, Dec. 2015, pp. 3088–3095.

[11] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proceedings - International Test Conference*. IEEE, 2008, pp. 1–10, DOI:10.1109/TEST.2007.4437639.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[13] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C. J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2014, pp. 1–6, DOI:10.1145/2593069.2593195.

[14] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Computer Safety, Reliability, and Security: 28th International Conference, SAFECOMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings*, B. Buth, G. Rabe, and T. Seyfarth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 283–296.

[15] M. Didehban and A. Shrivastava, "nZDC : A Compiler technique for near Zero Silent data Corruption," in *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. New York, New York, USA: ACM Press, 2016, pp. 1–6, DOI:10.1145/2897937.2898054.

[16] E. Chielle, R. S. Barth, Â. C. Lapolli, and F. L. Kastensmidt, "Configurable tool to protect processors against SEE by software-based detection techniques," in *LATW 2012 - 13th IEEE Latin American Test Workshop*. IEEE, Apr. 2012, pp. 1–6, DOI:10.1109/LATW.2012.6261259.

[17] C. Wang, H. S. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *International Symposium on Code Generation and Optimization, CGO 2007*. IEEE, Mar. 2007, pp. 244–256.

[18] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 25–36, 2000.

[19] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, Jun. 2007, pp. 544–553.

[20] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proceedings - International Symposium on Computer Architecture*. IEEE, 2005, pp. 148–159.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, p. 325, 2004.

[22] "Msp430fr59xx mixed-signal microcontrollers," Texas Instruments, Tech. Rep. SLAS704F, 2017.

[23] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault simulation and emulation tools to augment radiation-hardness assurance testing," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 2119–2142, Jun. 2013.

[24] R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection," in *IEEE Transactions on Nuclear Science*, vol. 47, no. 6 III, 2000, pp. 2405–2411.

[25] "Debug Server Scripting." [Online]. Available: http://software-dl.ti.com/ccs/esd/documents/users_guide/sdto_dss_handbook.html

[26] M. K. Bohman, "Compiler-assisted software fault tolerance for microcontrollers," Master's thesis, Brigham Young University, 2018.