COAST User Guide

Matthew Bohman, BYU

April 13, 2018

Contents

1 Introduction

This document serves to introduce the user to the COAST (COmpiler-Assisted Software fault Tolerance) tool. This tool leverages the LLVM compiler infrastructure to harden software against upsets in the compilation phase. For questions please email Jeff Goeders at jgoeders@byu.edu who has supervised this project. It is assumed that the user has read and set up the LLVM repository as outlined in the document "Getting Started with LLVM". This guide was developed using Ubuntu 16.04. For a more detailed explanation of COAST, as well as the reliability measurements of the different passes, please consult Matthew Bohman's Master's thesis (specific link coming soon, it will be posted at `https://scholarsarchive.byu.edu/etd/`).

2 COAST

COAST consists of a series of LLVM passes. The source code for these passes is found in the "llvm/projects" folder. This section covers the different passes available and their functions.

2.1 Building the Passes

Navigate to the "llvm/projects/build" folder and run `cmake ..` (note the two periods at the end of the command). This generates makefiles for each pass. Running `make` in this directory now builds all of the passes. They are represented as *.so files in this subdirectory.

2.2 Guide to the Passes

There are a number of different passes available. These are discussed more in detail below.

CFCSS This implements a form of Control Flow Checking via Software Signatures [1]. Basic blocks are assigned static signatures in compilation. When the code is executing it compares the current signature to the known static signature. This allows it to detect errors in the control flow of the program.

dataflowProtection This is the underlying pass behind the DWC and TMR passes.

debugStatements On occasion programs will compile properly, but the passes will introduce runtime errors. Use this pass to insert print statements into every basic block in the program. When the program is then run, it is easy to find the point in the LLVM IR where things went awry. Note that this incurs a **very** large penalty in both code size and runtime.

DWC This pass implements duplication with compare (DWC) as a form of data flow protection. DWC is also known as dual modular redundancy (DMR). It is based on EDDI [2]. Behind the scenes, this pass simply calls the dataflowProtection pass with the proper arguments.

exitMarker For software fault injection we found it helpful to have known breakpoints at the different places that `main()` can return. This pass places a function call to a dummy function, `EXIT_MARKER`, immediately before these return statements. Breakpoints placed at this function allow debuggers to access the final processor state.

TMR This pass implements triple modular redundancy (TMR) as a form of data flow protection. It is based on SWIFT-R [3] and Trikaya [4]. Behind the scenes, this pass simply calls the dataflowProtection pass with the proper arguments.

2.3 Using the Passes

Modifying source code in compilation requires a few steps. These steps refer to *clang*, *opt*, *llc*, and *lld*. These are found in the "llvm/build/bin" subdirectory. The commands below assume that this folder has been added to the user path. If you do not wish to do this, you must update the call to the program with the appropriate path. We recommend automating these steps using a Makefile.

1. Build LLVM
2. Build the passes as described in Section 2.1.
3. Compile your pass into LLVM IR (intermediate representation). If you are using C or C++ base files, this is done using Clang.
   ```
   clang -emit-llvm inFile.c -c -o outfile.clang.bc
   ```
4. Modify the IR using the desired passes. They must be loaded into *opt* before they can be used using the `-load` flag. Each pass must be loaded before it is used. Note that dataflowProtection must be included before DWC or TMR.
   ```
   opt -load <Necessary *.so file> <COAST flags> sourceFile.bc -o protectedFile.bc
   ```
5. Optional: turn the LLVM bytecode (.bc) into human-readable IR (.ll). This works for code pre- and post- modification.
   ```
   llvm-dis -f outFile.bc
   ```
6. Optional: Run the IR. This works for both IR files (.ll) and bytecode (.bc).
   ```
   lli inFile.ll
   ```
7. Compile the LLVM IR into assembly using *llc*. This can be passed into your favorite assembler. Alternately, the utility *lld* has recently been introduced to compile directly to common architectures. It is not included in this version of LLVM, but is a potential resource.
   ```
   llc -march=<target arch> <target options> protectedFile.bc -o outFile.s
   ```

Table 1: Pass 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 data on data stores (C4).
`-noStoreAddrSync`	Don't synchronize the address on data stores (C5).
`-ignoreFns=<X>`	<X> is a comma separated list of the functions that should not be replicated.
`-ignoreGlbls=<X>`	<X> is a comma separated list of the global variables that should not be replicated.
`-skipLibCalls=<X>`	<X> is a comma separated list of library functions that should only be called once.
`-replicateFnCalls=<X>`	<X> is a comma separated list of user functions where the body of the function should not be modified, but the call should be replicated instead.
`-configFile=<X>`	<X> is the path to the configuration file that has these options saved.
`-countErrors`	Enable TMR to track the number of errors corrected.
`-runtimeInitGlbls=<X>`	<X> is a comma separated list of the replicated global variables that should be initialized at runtime using `memcpy`.
`-i` or `-s`	Interleave (`-i`) the instruction replicas with the original instructions or group them together and place them immediately before the synchronization logic (`-s`). COAST defaults to `-s`.
`-dumpModule`	At the end of execution dump out the contents of the module to the command line. Mainly helpful for debugging purposes.
`-verbose`	Print out more information about what the pass is modifying.

3 Customization

In order to make COAST more flexible and useful, we have implemented a number of different options for the pass configurations. These can be called using the command line, included in a configuration file, or implemented using in-code directives. These options currently only modify DWC and TMR.

3.1 Command Line Options

The different options are listed and explained in Table 1. The first section of options in the table describes different options for instruction replication and synchronization, as used in previous work [5]. The next section allows for more fine-grained control of what should and should not be replicated. The final section describes a few other options available. These are described more in detail in this section.

3.1.1 Replication Rules

VAR3+, the set of replication rules introduced by Chielle et al. [5], instructs that all registers and instructions, except store instructions, should be duplicated. The data used in branches, the addresses before stores and jumps, and the data used in stores are all synchronized and checked against their duplicates. VAR3+ claims to catch 95% of data errors, so we used it as a starting point for automated mitigation. However, we

removed rule D2, which does not replicate store instructions, in favor of D1, which does. This results in replication of all variables in memory, and is desirable as microcontrollers have no guarantee of protected memory. The synchronization rules are included in both DWC and TMR protection. Rules C1 and C2, synchronizing before each read and write on the register, respectively, are not included in our pass because these were shown to provide an excessive amount of synchronization. G1, replicating all registers, and C6, synchronizing before branch or store instructions, cannot be disabled as these are necessary for the protection to function properly.

The first option, `-noMemReplication`, should be used whenever memory has a separate form of protection, such as error correcting codes (ECC). The option specifies that neither store instructions nor variables should be replicated. This can dramatically speed up the program because there are fewer memory accesses. Loads are still executed repeatedly from the same address to ensure no corruption occurs while processing the data.

The option `-noStoreAddrSync` corresponds to C5. In EDDI [2] memory was simply duplicated and each duplicate was offset from the original value by a constant. However, COAST runs before the linker, and thus has no notion of an address space. We implement rules C3 and C5, checking addresses before stores and loads, for data structures such as arrays and structs that have an offset from a base address. These offsets, instead of the base addresses, are compared in the synchronization logic.

3.1.2 Replication Scope

The user can specify any functions and global variables that should not be protected using `-ignoreFns` and `-ignoreGlbls`. At minimum, these options should be used to exclude code that interacts with hardware devices (GPIO, UART) from the SoR. Replicating this code is likely to lead to errors. The option `-replicateFnCalls` causes user functions to be called in a coarse grained way, meaning the call is replicated instead of fine-grained instruction replication within the function body. Library function calls can also be excluded from replication via the flag `-skipLibCalls`, which causes those calls to only be executed once. These two options should be used when multiple independent copies of a return value should be generated, instead of a single return value propagating through all replicated instructions. Changing the scope of replication can cause problems across function calls. Table 2 shows the proper use of these options.

3.1.3 Other Options

Error Logging This option was developed for tests in a radiation beam, where upsets are stochastically distributed, unlike fault injection tests where one upset is guaranteed for each run. COAST can be instructed to keep track of the number of corrected faults via the flag `-countErrors`. This flag allows the program to detect corrected upsets, which yields more precise results on the number of radiation-induced SEUs. This option is only applicable to TMR because DWC halts on the first error. A global variable, `TMR_ERROR_CNT`, is incremented each time that all three copies of the datum do not agree. If this global is not present in the source code then the pass creates it. The user can print this value at the end of program execution, or read

Table 2: When to use replication command line options

Desired Behavior	Function Type	Option	Use Case
Protect called function	User	Default	Standard behavior, use for most cases.
	Library	N/A	Cannot modify library calls. Instead, see the case below.
Replicate call	User	`-replicateFnCalls=<X>`	When the return value needs to be unique to each instruction replica, e.g. pointers.
	Library	Default	By default the library calls are performed repeatedly. Use for most calls.
Call once, unmodified	User	`-ignoreFns=<X>`	Interrupt service routines and synchronization logic, such as polling on an external pin.
	Library	`-skipLibCalls=<X>`	Whenever the call should not be repeated, such as calls interfacing with I/O.

it using a debugging tool.

Error Handlers The user has the choice of how to handle DWC and CFCSS errors because these are uncorrectable. The default behavior is to create `abort()` function calls if errors are detected. However, user functions can be called in place of `abort()`. In order to do so, the source code needs a definition for the function `void FAULT_DETECTED_DWC()` or `void FAULT_DETECTED_CFCSS` for DWC and CFCSS, respectively.

Input Initialization Global variables with initial values provide an interesting problem for testing. By default, these initial values are assigned to each replicate at compile time. This models the scenario where the SoR expands into the source of the data. However, this does not accurately model the case when code inputs need to be replicated at runtime. This could happen, for instance, if a UART was feeding data into a program and storing the result in a global variable. When global variables are listed using `-runtimeInitGlbls` the pass inserts `memcpy` calls to copy global variable data into the replicates at runtime. This supports scalar values as well as aggregate data types, such as arrays and structures.

Interleaving In previous work [2] replicated instructions have all been placed immediately after the original instructions. Interleaving instructions in this manner effectively reduces the number of available registers because each load statement executes repeatedly, causing each original value to occupy more registers. For TMR, this means that a single load instruction in the initial code uses three registers in the protected program. As a result, the processor may start using the stack as extra storage. This introduces additional memory accesses, increasing both the code size and execution time. Placing each set of replicated instructions

immediately before the next synchronization point lessens the pressure on the register file by eliminating the need for multiple copies of data to be live simultaneously.

By default, COAST groups copies of instructions before synchronization points, effectively partitioning regions of code into segments where each copy of the program runs uninterrupted. Alternately, the user can specify that instructions should be interleaved using `-i`.

Printing Status Messages Using the `-verbose` flag will print more information about what the pass is doing. This includes removing unused functions and unused global strings. These are mainly helpful for examining when your code is not behaving exactly as expected.

If you are developing passes, then on occasion you might need to include more printing statements. Using `-dumpModule` causes the pass to print out the entirety of the LLVM module to the command line in a format that can be tested using *lli*. This is mainly helpful if the pass is not cleaning up after itself properly. The function `dumpModule()` can also be placed in different places in the code for additional debugging capabilities.

3.2 Configuration File

Alternately, instead of repeating the same command line options across several compilations, we have created a configuration file, "functions.config" that can capture the same behavior. It is found in the `dataflowProtection` pass folder. The location of this file can be specified using the `-configFile=<...>` option. The options are the same as the command line alternatives.

3.3 In-code Directives

Table 3: In-line code directives.

Directive	Effect
`__DEFAULT_xMR`	Include at the top of the code. Set the default processing to be to replicate every piece of code except those specifically tagged. This is the default behavior.
`__NO_xMR`	Set the default behavior of COAST to not replicate anything except what is specifically tagged.
`__NO_xMR`	Used to tag functions and variables that should not be replicated. Functions tagged in this manner behave as if they were passed to `-ignoreFns`.
`__xMR`	Designate functions and variables that should be cloned. This replicates function bodies and modifies the function signature.
`__xMR_FN_CALL`	Available for functions only. The same as `replicateFnCalls` above. Repeat function calls instead of modifying the function body.

As an alternative to the configuration file or the command line option, COAST also supports in-code annotations of the same options. These are implemented as compiler annotations to prevent CLANG from optimizing them away. Macros which encapsulate these annotations are available in "COAST.h" located in the "dataflowProtection" folder. Including this header file enables the use of the directives. The available

options are shown in Table 3. The first two entries describe the default behavior of the pass. The remainder are specific directives for variables and functions.

As before, these instructions only apply to data flow protection. The protections are equally applicable for both DWC/DMR and TMR. Figure 1 contains an example of how these directives can be used.

```
#include ''COAST.h''

//By default, apply xMR to everything
__DEFAULT_xMR

//globalCnt is not replicated
int __NO_xMR globalCnt;

//The body of initialize is replicated
void __xMR initialize(){
    //localVariable is excluded from the replication
    int __NO_xMR localVariable;
    ...
}

//printStatus is only executed once and not modified
int __NO_xMR printStatus(){
    ...
}

//useMalloc is called repeatedly, but the function is not modified
int* __xMR_FN_CALL useMalloc(){
    ...
}

void main(){
    initialize();
    useMalloc();
    printStatus();
}
```

Figure 1: Example of how to use in-code directives.

4 Troubleshooting

Although it is unlikely, there is a possibility that COAST could cause user code to crash. This is most often due to complications over what should be replicated, as described in Section 3.1.2. If the crash occurs during compilation, please submit a report to jgoeders@byu.edu. If the code compiles but does not run properly, here are several steps we have found helpful. Note that running with DWC often exposes these errors, but TMR silently masks incorrect execution, which can make debugging difficult.

- Check to see if the program runs using *lli* before and after the optimizer, then test if the generated

binary runs on your platform. This allows you to test that *llc* is operating properly.

- You cannot replicate functions that are passed by reference into library calls. This may or may not be possible in user calls. Use `-ignoreFns` for these.
- For systems with limited resources, duplicating or triplicating code can take up too much RAM or ROM and cause the processor to halt. Test if a smaller program can run.
- The majority of bugs that we have encountered have stemmed from incorrect usage of customization. Please refer to Table 2 and ensure that each function call behaves properly. Many of these bugs have stemmed from user wrappers to `malloc` and `free`. The call was not replicated, so all of the instructions operated on a single piece of data, which caused multiple `free` calls on the same memory address.
- Another point of customization to be aware of is how to handle hardware interactions. Calls to hardware resources, such as a UART, should be marked so they are not replicated unless specifically required.
- Be aware of synchronization logic. If a variable changes between accesses of instruction copies, then the copies will fail when compared.
- Use the `-debugStatements` flag to explore the IR and find the exact point of failure.

References

[1] 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.

[2] ——, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[3] 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.

[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] 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.