COAST

Oct 29, 2020

Contents:

COmpiler-**A**ssisted **S**oftware fault **T**olerance

CHAPTER 1

Getting Started

1.1 What is LLVM?

For a good introduction to LLVM, please refer to http://www.cs.cornell.edu/~asampson/blog/llvm.html

1.2 Prerequisites

- Have a version of Linux that has `cmake` and `make` installed.

For reference, development of this tool has been done on Ubuntu 16.04 and 18.04.

1.3 Installing LLVM

There are a few different ways that LLVM and Clang can be installed, depending on your system and preferences. This project uses LLVM v7.0, so make sure you install the correct version.

1.3.1 Option 1 - System Packages

With Ubuntu 18.04 and higher, use the following commands:

```
sudo apt install llvm-7
sudo apt install clang-7
```

Other Linux distributions may also have packages available.

1.3.2 Option 2 - Precompiled Binaries

You can obtain precompiled binaries from the official GitHub page for the LLVM project.

1.3.3 Option 3 - Build from Source

If the other two options do not work for your system, or if you prefer to have access to the source files for enhanced debugging purposes, you can build LLVM from source.

- Create a folder to house the repository. It is recommended that the folder containing this repository be in your home directory. For example, `~/coast/`.
- Check out the project:

```
git clone https://github.com/byuccl/coast.git ~/coast
```

- Change to the "build" directory and configure the Makefiles. Example invocation:

```
cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug -DLLVM_ENABLE_ASSERTIONS=On ../
↪llvm-project/llvm/
```

To enable support for RISCV targets, add `-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=RISCV` to the `cmake` invocation.

See the `README.md` in the "build" folder for more information on how to further configure LLVM.

- Run `make`. This may take quite a while, up to an hour or more if you do not parallelize the job. Adding the flag `-jn` allows you to parallelize across `n` cores.

Note: The higher the number the faster the builds will take, but the more RAM will be used. Parallelizing across 7 cores can take over 16 GB of RAM. If you run out of RAM, the compilation can fail. In this case simply re-run `make` without any parallelization flags to finish the compilation.

If you wish to add the LLVM binaries to your `PATH` variable, add the following to the end of your `.bashrc` file:

```
export PATH="/home/$USER/coast/build/bin:$PATH"
```

1.4 Building the Passes

To build the passes so they can be used to protect your code:

- Go the "projects" directory
- Make a new subdirectory called "build" and `cd` into it
- Run `cmake ..`
- Run `make` (with optional `-jn` flag as before)

CHAPTER 2

Using the Makefile System

We have provided a set of Makefiles that can be used to build the benchmarks in the "tests" folder. They are conditionally included to support building executables for various platforms without unnecessary code replication.

2.1 Targets

There are two Make targets commonly used by all of the Makefiles. The first is `exe`, which builds the executable itself. The second is `program`, which runs the executable. If the target architecture is an external device, it will upload the file to the device. If it is a local architecture, such as `lli` or `x86`, then it will run on the host machine. Some architectures incorporate FPGAs, and so have an additional Make target called `configure`, which will upload a bitstream to the FPGA.

2.2 Extending the Makefile system

Adding support for additional platforms requires a new Makefile be created that contains the build flow for the target platform. The basic idea is to

1. Compile the source code to LLVM IR
2. Run the IR through `opt` (and enable the `-DWC` or `-TMR` passes as necessary)
3. Link the COAST protected code with any other object code in the project
4. Assemble to target machine language

2.3 Example

A good example to look at is the pseudo target `lli`. This is LLVM's target independent IR source interpreter. It can execute `.ll` or `.bc` files (plain-text IR or compiled bytecode). It is fairly simple because it does not require an

assembly step. For an example of converting from the protected IR to machine code, look at the Makefile for compiling to the Pynq architecture.

CHAPTER 3

Passes

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

3.1 Description

- **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.

[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.

- **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.
- **smallProfile**: This pass can be used to collect dynamic function call counts.

3.2 Configuration Options

COAST can be configured to apply replicating rules in other ways than by the default using *Command Line Parameters*, *In-code Directives*, and a *Configuration File*.

3.2.1 Command Line Parameters

These options are only applicable to the `-DWC` and `-TMR` passes.

The details for each of these options can be found in the *Details* section.

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).
`-storeDataSync`	Force synchronizing data on data stores (C4).

`-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.

[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.

-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.

Note: Replication rules defined by Chielle et al.[5].

New in version 1.4.

-isrFunctions=<X>	<X> is a comma separated list of the function names that should be treated as Interrupt Service Routines (ISRs).
-cloneReturn=<X>	<X> is a comma separated list of the function names that should have their return values cloned.
-cloneAfterCall=<X>	<X> is a comma separated list of the function names that will have their arguments cloned after the call.
-protectedLibFn=<X>	<X> is a comma separated list of the function names that should be protected without having their signatures changed.
-countSyncs	Instructs COAST to keep track of the dynamic number of synchronization checks. Requires -countErrors.
-protectStack	Enable experimental stack protection.
-noCloneOpsCheck	Disable exiting on failure of check verifyCloningSuccess.

3.2.2 In-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.
__DEFAULT_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.

New in version 1.2.

[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.

`__COAST_VOLATILE`	Used to mark global variables as ones that the pass should not remove, even if it does not appear to be used.
`__COAST_IGNORE_GLOBAL(name)`	Ignore checks for global variable replication in function following this directive. See section *Replication Scope*.

`MALLOC_WRAPPER_REGISTER(fname)`	Give the name of a `malloc()`-like function that will be replicated. Should be treated the same as a function prototype.
`MALLOC_WRAPPER_CALL(fname, x)`	Make a call to the function registered using the above macro. This will be replicated by COAST, using the clones of the arguments.
`PRINTF_WRAPPER_REGISTER(fname)`	Give the name of a `printf()`-like function that will be replicated. Should be treated the same as a function prototype.
`PRINTF_WRAPPER_CALL (fname, fmt, ...)`	Make a call to the function registered using the above macro. This will be replicated by COAST, using the clones of the arguments.
`GENERIC_COAST_WRAPPER(fname)`	Make your own wrapper function for COAST to replicate calls to. Used in both declaring and calling the function.

New in version 1.4.

`__ISR_FUNC`	Used to mark functions that should be treated as Interrupt Service Routines (ISRs).
`__xMR_RET_VAL`	Used to mark functions that should have their return values cloned.
`__xMR_PROT_LIB`	Used to mark functions that should be protected without having their signatures changed.
`__xMR_ALL_AFTER_CALL`	Used to mark functions that should have their arguments cloned after the call.
`__xMR_AFTER_CALL(fname, x)`	Specific version of the above macro. Specifiy the arg numbers as `(name, 1_2_3)`. Must be registered, similar to `GENERIC_COAST_WRAPPER(fname)`
`__NO_xMR_ARG(num)`	The argument [num] should not be replicated. If multiple arguments need to be marked, this directive should be placed on the function multiple times.
`__COAST_NO_INLINE`	Convenience for no-inlining functions

See the file COAST.h

3.2.3 Configuration File

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.

The default file contains functions we have identified as commonly treated differently than the default COAST options.

3.2.4 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.
Protect without changing signature	User	`-protectedLibFn=<X>`	Library functions you have the source code for.
	Library	N/A	Can't protect it if you don't have the source code.
Return multiple values	User	`-cloneReturn=<X>`	When calling the function multiple times would have unwanted side effects.
	Library	N/A	Cannot modify the source code of library functions.

3.3 Details

3.3.1 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, 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.

Changed in version 1.2.

As of the October 2019 release, COAST no longer syncs before storing data. Test data indicated that, in many cases, the number of synchronization points generated by this rule limited the effective protection that the replication of variables afforded. This behavior can be overridden using the -storeDataSync flag.

3.3.2 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.

New in version 1.2.

Before processing the IR code, COAST begins by checking to make sure the replication scope rules it was given are consistent. It checks to make sure all cloned globals are only used in functions that are also protected. If they are not, the compilation will fail, with an error message informing the user which global is used in which function. The user has the option to ignore these checks if they feel that it is safe. This is done using the __COAST_IGNORE_GLOBAL macro mentioned above.

New in version 1.4.

There are also some options that have been added that allow more fine-grained control over how different functions and values are protected. The first of these is the command line argument -cloneReturn, or directive __xMR_RET_VAL. This instructs COAST that the return value of the function should be cloned. This has been implemented by adding extra arguments to the end of the parameter list that are pointer types of the normal return value. This prevents the values from passing through a bottleneck. This is particulary useful for functions that return addresses to memory spaces that have been dynamically allocated.

Another recently added option is the ability to mark functions as "protected library functions" (-protectedLibFn=<X>, __xMR_PROT_LIB). The idea behind this is that there are some functions that should not have their signatures changed, but should still have their bodies protected.

Another interesting feature added in this version is the ability to copy the value of the original variable into its clone(s) *after* the function call has been completed. An example of when this might be useful is the function sscanf. This function will read values from a string based on a format specifier and put the values into the pointers provided.

```
sscanf (sentence,"%s %*s %d",str,&i);
```

This will allow the copies of the variables to stay in sync with each other even when calling a library function that can only be called once, that modifies a variable by reference.

We have introduced a way to mark functions as Interrupt Service Routines (ISRs), which means they will not be changed in any way, nor removed if they don't appear to have any uses.

COAST now has much better support for changing the protection of variables that are local to protected functions. They can be excluded from the Scope of Replication using the macro __NO_xMR. Even function arguments can be

excluded using the macro `__NO_xMR_ARG(num)`.

3.3.3 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 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 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.

If you are developing passes, then on occasion you might need to include more printing statements. Using the `-dumpModule` flag causes the pass to print out the entirety of the LLVM module to the command line in LLVM IR format.

3.4 Debugging Tools

3.4.1 COAST verbose output

As mentioned above, COAST supports the `-verbose` and `-dumpModule` flags. The `-verbose` output lists alls of the in-code directives processed, which functions are having their signatures changed, as well as any unused globals or functions being removed. COAST will also print warnings or errors about unsupported language constructs being used.

Using the `-dumpModule` flag is useful to get an idea of what COAST is doing if it's failing to finish compilation. The function `dumpModule()` can also be placed in different places in the code for additional debugging capabilities.

Since the module will be output to the `stderr` stream, and it can be quite a lot of data, it is important to redirect the output properly.

Example: `opt -TMR -dumpModule input.bc -o output.bc > dump.ll 2>&1`

3.4.2 Debug Statements

By default, the Debug Statements pass will add code to the beginning of every basic block that prints out the function name followed by the name of the basic block. For example, you would expect the first message to be `main->entry`. This can produce 100s of MegaBytes of data, so it is important to redirect this output to a file, as shown in the example above. This verbose output represents a complete call graph of the execution, although trawling through all of this data can be quite difficult.

New in version 1.2.

There is an option to only add print statements to certain functions. Pass `-fnPrintList=` with a comma-separated list of function names that will be instrumented with the print statements. This will allow examining smaller parts of the execution at a time.

3.4.3 Small Profiler

New in version 1.2.

The Small Profiler is a pass which simply counts the number of calls to each function in the module. It creates global variables that correspond to each function in the module. Each time a function is called, the corresponding global variable is incremented. The pass adds a call to a function named `PRINT_PROFILE_STATS` immediately before the `main` function exits. If the program does not terminate, calls to this function may be inserted manually by the programmer.

This pass also has two command line parameters:

Command line option	Effect
`printFnName`	The name of the function that is used to print the stats. The default is `printf`. This flag is for if the platform does not support `printf`.
`noPrint`	Do not insert the call to `PRINT_PROFILE_STATS`.

CHAPTER 4

Scope of Replication

We use the term Sphere of Replication (SoR) to indicate which portions of the source code are to be protected. In large applications, it may be too much overhead to have the entire program protected by COAST, so there is a way to configure COAST to only protect certain functions, using macros found in the header file COAST.h.

4.1 Configuration

COAST allows for very detailed control over what belongs inside or outside of the Scope of Replication. There are numerous *Command Line Parameters* and *In-code Directives* which allow for projects to be configured very precisely. COAST even includes a verification step that tries to ensure all SoR rules are self-consistent. It can detect if protected global variables are used inside unprotected functions, or vice-versa. However, this system is not perfect, and so the application writer must be aware of the potential pitfalls that could be encountered when using specific replication rules.

4.2 Pointer Crossings

One of the most common problems to be aware of is pointers which cross the SoR boundaries. Many applications use dynamically allocated memory. If the function that allocates this memory is inside the SoR, then *all* references to these addresses must also be within the SoR. It is true that read-only access would not cause errors, as in the case of using `printf` to view the value of such a pointer. But no writes can happen outside the SoR, otherwise the addresses will get out of sync.

4.3 Example

The unit test linkedList.c shows exactly how SoR crossings can go wrong by looking at a possible implementation of a linked list.

CHAPTER 5

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 the *When to use replication command line options* and *Replication Scope* sections. If the crash occurs during compilation, please submit a report to jgoeders@byu.edu or create an issue. 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.

5.1 Troubleshooting Ideas

- 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 *When to use replication command line options* 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, such as volatile hardware registers, then the copies will fail when compared.
- Use the `-debugStatements` flag to explore the IR and find the exact point of failure. See the *Debugging Tools* section for more information.

- You may get an error that looks something like `undefined symbol: ZTV18dataflowProtection` when you try to run DWC or TMR. This occurs when you do not load the dataflowProtection pass before the DWC or TMR pass. Include `-load <Path to dataflow protection.so>` in your call to `opt`.
- If compiling a C++ project, be aware that the compiler will often mangle the names of functions. In this case, the function names passed in to COAST may need to be changed. Examine the LLVM IR output being given to `opt` to make sure they are correct.

CHAPTER 6

Release Notes

6.1 v1.5 - October 2020

6.1.1 Fault Injection Supervisor

Python scripts which comprise the *Fault Injection* interface.

6.1.2 FreeRTOS Example Applications

Example *FreeRTOS Applications* that run on the FreeRTOS kernel, plus how to protect them with COAST.

Documentation also updated to include information about the *Baremetal Benchmarks*.

6.2 v1.4 - August 2020

6.2.1 Features

- Support for cloning function return values
- New unit tests
- Better copying of debug info
- Experimental stack protection
- 7 new command line arguments
  See *Command Line Parameters* for more information.

6.2.2 Directives

7 new directives

- `__ISR_FUNC`
- `__xMR_RET_VAL`
- `__xMR_PROT_LIB`
- `__xMR_ALL_AFTER_CALL`
- `__xMR_AFTER_CALL`
- `__NO_xMR_ARG`
- `__COAST_NO_INLINE`

See *In-code Directives* for more information.

6.2.3 Bug Fixes

- Correct support for variadic functions
- Fix up debug info for global variables so it works better with GDB
- Better removal of unused functions
- Official way of marking ISR functions instead of function name text matching

6.3 v1.3 - November 2019

Changed the source of the LLVM project files from SVN (deprecated) to the Git mono-repo, version 7.1.0.

6.4 v1.2 - October 2019

6.4.1 Features

- Support for `invoke` instructions.
- Replication rules, does NOT sync on stores by default, added flag to enable turning that on (`-storeDataSync`).
- Support for compiling multiple files in the same project at different times (using the `-noMain` flag).
- Before running the pass, validates that the replication rules given to COAST are consistent with themselves.
- Can sync on vector types.
- Added more unit tests, along with a test driver.

6.4.2 Directives

- Added directive `__SKIP_FN_CALL` that has the same behavior as `-skipFnCalls=` command line parameter.
- Can add option to not check globals crossing Sphere of Replication (`__COAST_IGNORE_GLOBAL(name)`).

- Added directive macro for marking variables as volatile.
- Treats any globals or functions marked with `__attribute__((used))` as volatile and will not remove them. Also true for globals used in functions marked as "used".
- Added wrapper macros for calling a function with the clones of the arguments. Useful for `printf()` and `malloc()`, etc, when you only want specific calls to be replicated.

6.4.3 Bug Fixes

Thanks to Christos Gentsos for pointing out some errors in the code base.

- Allow more usage of function pointers by printing warning message instead of crashing.
- Added various missing `nullptr` checks.
- Fixed crashing on some `void` return type functions.
- Better cleanup of stale pointers.

6.4.4 Debugging Tools

- Added an option to the `DebugStatements` pass that only adds print statements to specified functions.
- Created a simplistic profiling pass called `SmallProfile` that can collect function call counts.
- Support for preserving debug info when source is compiled with debug flags.

CHAPTER 7

Using an IDE to aid LLVM development

We have used both Eclipse and Visual Studio Code in the development of COAST. This is very helpful because it allows code completion hints that inform you what methods are available for specific classes.

7.1 Using Eclipse with LLVM

This guide was written for Eclipse 4.10.0 using the CDT.

7.1.1 Setting up the project

1. Select "File -> New -> Makefile Project with Existing Code".
2. Enter `projects` as the project name.
3. For the existing code location field, browse to the projects directory
4. Use the "Linux GCC" toolchain.
5. Right click on your project directory and select "Properties"
6. Navigate to "C/C++ Build" and change the build directory to your `projects/build` folder using the "File system" button.
7. Change to the "Behavior" tab and enable parallel builds. We recommend using 3-4 parallel jobs.
8. Click "Apply" then "Apply and Close".
9. When you click on the "Build" button the projects will be compiled.

7.1.2 Building the projects

1. Right click on the `projects/build` subdirectory, then "Make Targets -> Create".
2. Call the target name `all` and click OK.

3. To build your pass, right click on the build folder and click “Make Targets -> Build -> Build” (with the target `all` selected).
4. After the first time that you’ve done this, you can rebuild all your passes by pressing `F9`.

7.1.3 Fixing the CDT settings

The default settings of the project are not sufficient to allow the Eclipse CDT indexer to work correctly. While not necessary to fix the CDT settings, it allows you to use the auotcomplete functionality of Eclipse.

1. Right-click on the project and select “Properties”
2. Under “C/C++ General” select “Paths and Symbols”
3. Add a new Include Directory using the “Add” button
4. Select “File System”
5. Navigate to the repository root, then select `llvm/include`
6. Check the box “Add to all languages,” then click “OK”
7. On the left pane, select “Preprocessor Include Paths, Macros, etc”
8. On the “Providers” select “CDT GCC Built-in Compiler Settings”
9. Edit the “Command to get compiler specs” by putting `std=c++11` right before `${INPUTS}`
10. Move the entry “CDT GCC Built-in Compiler Settings” to the top of the list using the “Move Up” button
11. Select “Apply and Close”

1. Select “Window” -> “Preferences”
2. Select “C/C++” -> “Build” -> “Settings”
3. Under the “Discovery” tab select “CDT GCC Built-in Compiler Settings”
4. Edit the “Command to get compiler specs” the same as before
5. Select “Apply and Close”

7.2 Using VS Code with LLVM

1. Open VS Code
2. File -> Open Folder
3. Select the directory that contains the files for the pass you want to develop
4. On the bottom ribbon at the right there will be a button next to the language configuration (ours says “Linux”)
5. Hovering over this button says “C/C++ Configuration”. Click on it
6. You will be taken to a page that allows you to set up a specific configuration for this directory.
7. Click the button “Add Configuration” and give it a name
8. Add the path to the LLVM include files in the section “Include path”
9. For example, because I built LLVM from source, I added the following:

```
/home/$USER/coast/llvm-project/llvm/include
/home/$USER/coast/build/include
```

CHAPTER 8

Control Flow Checking via Software Signatures (CFCSS)

8.1 Introduction

As part of our research into software error mitigation, we recognized the necessity of checking for control flow errors along with dataflow errors.

8.2 Previous Work

Control-flow checking by software signatures[1]

8.3 Algorithm

The algorithm we determined to use is one found in the research paper mentioned above. A brief description will be included here.

A program may be split into a representation using "basic blocks." A basic block (b_n) is a collection of sequential instructions, into which there is only one entry point, and out of which there is only one exit point. Many basic blocks may branch into a single basic block, and a single basic block may branch out to many others. The process of ensuring that these transitions between basic blocks are legal is called Control Flow Checking. A legal transition is defined as one that is allowed by the control flow graph determined at compile time before the program is run.

At compile time, a graph is generated showing all legal branches. Each basic block is represented by a node. A unique signature (s_n) is assigned to each basic block. Along with this, a signature difference (d_n) is assigned to each basic block, which is calculated by taking the bit-wise XOR ($\oplus$) of the current block and its successor. When the program is run, a run-time signature tracker (G_n) is updated with the signature of the current basic block. When the program branches to a new basic block, the signature tracker is XOR'd with the signature difference of the new block:

[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.

$$G_n \oplus s_n = d_n$$

Because the XOR operation can undo itself, the result should equal the signature of the current block. If it does not, then a control flow error has been detected.

Fig. 8.1: correct vs incorrect branching

8.3.1 Branch Fan-in

There is a danger when dealing with dense control flow graphs that there will be a configuration as seen in Fig. 8.2

Fig. 8.2: branch fan-in problem

If b_1 and b_3 are assigned the same signature, then there will be no issue branching to b_4 . However, this opens up the possibility for illegal branching from b_1 to b_5 without being caught. If all signatures are generated randomly, without any duplicates, then b_4 will register correct branching from either b_1 or b_3 , but not both.

This necessitates the addition of the run-time signature adjuster. D_n This is an additional number that is calculated at compile time for each basic block, then updated as the program executes. It is used to adjust for the differences created by this branch fan-in problem.

Fig. 8.3: run-time signature adjuster

In the case of the branch from b_1 to b_4 , the signature adjuster will be 0. In the case of the branch from b_3 to b_4 , the signature adjuster will be

$$D_3 = s_3 \oplus d_4 \oplus s_4$$

such that

$$G_4 = G_3 \oplus d_4 \oplus D_3$$

8.4 Modifications

Although the algorithm described above is very robust, there were some instances where it does not perform correctly. If a node has two successors which are themselves both branch fan-in nodes (as in Fig. 8.4), the algorithm will correctly assign a signature adjuster value for one branch, but not for the other.

To solve this problem, we determined to insert an extra basic block to act as a buffer. This would go between the predecessor with the invalid signature adjuster and the successor that is the branch fan-in node (see Fig. 8.6) It would contain no instructions other than those that verify proper control flow. Because this buffer block would only have one predecessor, it would not need to use the signature adjuster, whatever the value might be. The value for D_8 for the buffer block would be determined to allow correct branching to the successor node.

8.5 Implementation

We implemented this algorithm using LLVM. It was implemented as a pass that the optimizer runs before the back-end compiles the assembly into machine code. This particular implementation worked very well with the algorithm, because LLVM automatically splits its programs into basic blocks. One of the challenges this presented was compiling

Fig. 8.4: multiple successors with branch fan-in

Fig. 8.5: run-time signature adjuster error

for a 16-bit microprocessor. In order to save space, the signatures were generated as unsigned 16-bit numbers. This gives 65,535 possible signatures to use, which far surpasses the number of basic blocks you could fit in such a small memory space as we had on our device.

To deal with the multiple fan-in successor problem mentioned above, we ran the signature generation step as normal. Then we checked the entire graph to see if there were any mismatched signatures. If there were, we inserted a buffer block to deal with that problem and updated the surrounding blocks to match the new block.

To implement the control flow checking, we inserted a set of instructions at the beginning of each basic block to do the XOR operation specified above. We also inserted instructions at the end of each block to update the run-time signature tracker to be the signature of the block about to be left.

One of the optimizations we used was to only insert the extra XOR operation when D_n1 was $\neq 0$. This is one reason why the buffer block fix worked.

8.6 Notes

This pass was created for the purposes of studying LLVM IR and the LLVM C++ framework. It is not actively being maintained.

Fig. 8.6: using the buffer block

Fig. 8.7: inserting instructions into basic blocks

CHAPTER 9

Tests

9.1 Baremetal Benchmarks

In the course of developing COAST, it became necessary to validate that COAST-protected code operates as expected. We have collected a number of benchmarks to put COAST through different use cases. Some of these can be run on Linux, and others have been built to target a specific architecture.

Some of the tests are from known test suites, adapted to work with COAST. Others are of our own concoction. The tests are found in the repo in this directory We list some noteworthy directories below:

- aes - An implementation of AES, borrowed from this repo along with `cache_test`, `matrixMultiply`, and `qsort`.
- chstone - adapted from CHStone test suite.
- makefiles - the backbone of the testing setup, this directory has all of the files for configuring GNU Make to run the tests.
- TMRregression/unitTests - Small unit tests which test very specific COAST functionality. Corner cases usually uncovered when trying to protect larger applications. The directory `TMRregression` contains scripts for running these and other tests.

9.2 FreeRTOS Applications

Protecting a FreeRTOS kernel and application is much more complex a task than protecting a baremetal program. The files can be found here, and the COAST configuration needed to get the applications to work is detailed in the Makefile.

CHAPTER 10

Fault Injection

To supplement the testing done in actual high-radiation environments, we have developed a system to inject faults into the applications we want to test. This system is built on QEMU, the Quick EMUlator. We currently support the ARM Cortex-A9 processor, the main processing unit found in the Zynq-7000 SoC, a part we have often used in radiation tests.

The basic idea is to have a QEMU instance running the application that also runs a GDB stub. Using the GDB interface, we can change values in the memory or registers as desired. We utilize a QEMU plugin to keep track of exactly how cycles have elapsed so that the faults injected can be distributed evenly through time.

Instructions for building QEMU and the associated plugins can be found in the README of our QEMU fork.

Instructions for using the fault injector can be found by executing

```
python3 supervisor.py -h
```

in the directory `coast/simulation/platform`.

CHAPTER 11

Folder guide

11.1 boards

This folder has support files needed for the various target architectures we have used in testing COAST.

11.2 build

This folder contains instructions on how to build LLVM, and when built will contain the binaries needed to compile source code. Note: building LLVM from source is optional.

11.3 projects

The passes that we have developed as part of COAST.

11.4 rtos

Example applications for FreeRTOS and how to use it with COAST.

11.5 simulation

Files for running fault injection campaigns.

11.6 tests

Benchmarks we use to validate the correct operation of COAST.

CHAPTER 12

Results

See the results of fault injection and radiation beam testing

12.1 MSP430

The current results are shown below. Detailed descriptions of the benchmarks, methodology, and analysis of the results are available in Matthew Bohman's Master's thesis.

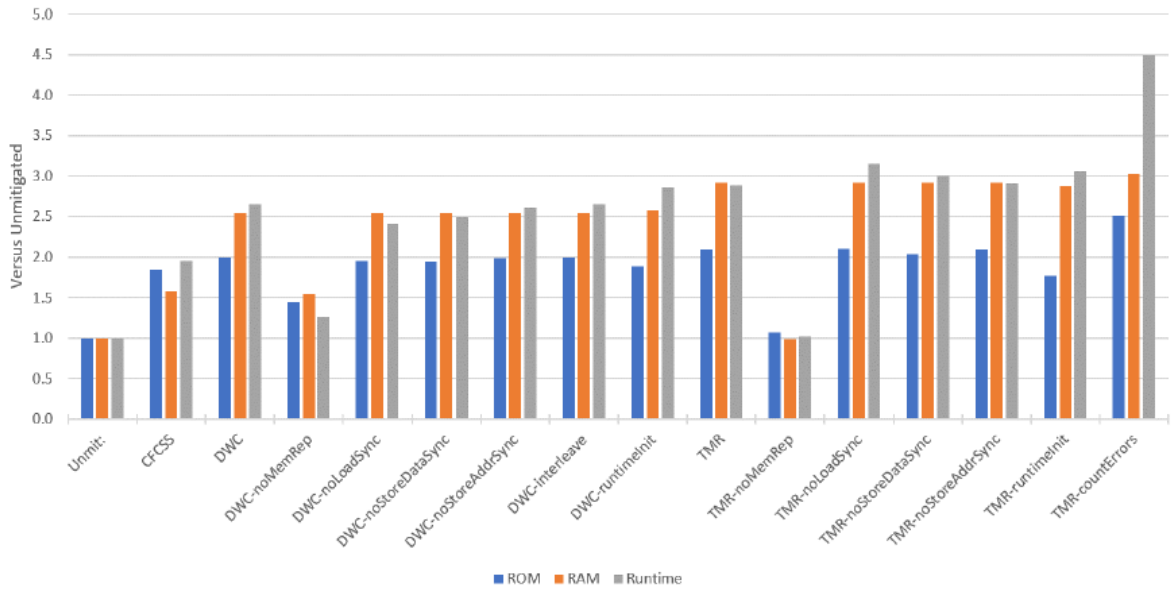
Option	ROM Usage	RAM Usage	Runtime	MWTF	Fault Coverage
Unmitigated	1.0x	1.0x	1.0x	1.0x	85.4%
-CFCSS	1.8x	1.6x	2.0x	0.8x	87.9%
-DWC	2.0x	2.5x	2.6x	18.6x	99.0%
-noMemReplication	1.5x	1.5x	1.3x	1.0x	85.8%
-noLoadSync	2.0x	2.5x	2.4x	28.3x	99.1%
-noStoreDataSync	1.9x	2.5x	2.5x	21.6x	99.0%
-noStoreAddrSync	2.0x	2.5x	2.6x	26.7x	99.1%
-i	2.0x	2.5x	2.6x	24.6x	99.3%
-runtimeInitGlobals	1.9x	2.6x	2.9x	23.2x	99.4%
-TMR	2.1x	2.9x	2.9x	21.6x	98.8%
-noMemReplication	1.1x	1.0x	1.0x	1.1x	86.3%
-noLoadSync	2.1x	2.9x	3.2x	16.8x	98.9%
-noStoreDataSync	2.0x	2.9x	3.0x	18.0x	98.9%
-noStoreAddrSync	2.1x	2.9x	2.9x	18.5x	98.8%
-runtimeInitGlobals	1.8x	2.9x	3.1x	16.7x	98.9%
-countErrors	2.5x	3.0x	4.5x	2.3x	90.7%

Table 4.2: Fault Injection Results

Option	MxM			CRC			QS		
	OK	Err	DWC*	OK	Err	DWC*	OK	Err	DWC*
Unmitigated	4202	798	–	4329	671	–	4282	718	–
-CFCSS	3227	664	1109	3316	544	1140	2107	602	2291
-DWC	3310	25	1665	3768	99	1133	3690	25	1285
-noMemReplication	4210	790	0	4324	675	1	3643	666	8
-noLoadSync	3349	11	1640	3722	96	1182	3643	29	1328
-noStoreDataSync	3475	19	1506	3655	99	1246	3629	29	1346
-noStoreAddrSync	3394	13	1593	3671	94	1235	3666	25	1309
-i	3380	17	1603	3739	73	1188	3690	22	1288
-runtimeInitGlobals	3302	20	1678	–	–	–	3666	41	1293
-TMR	4978	22	–	4905	95	–	4940	60	–
-noMemReplication	4238	762	–	4370	630	–	4338	662	–
-noLoadSync	4962	38	–	4924	76	–	4947	53	–
-noStoreDataSync	4966	34	–	4924	76	–	4949	51	–
-noStoreAddrSync	4967	33	–	4899	101	–	4957	43	–
-runtimeInitGlobals	4955	45	–	–	–	–	4934	66	–
-countErrors	4749	251	–	4318	682	–	–	–	–

Figure 4.3: Averaged fault injection results

(a) Overhead in ROM, RAM, and runtime



(b) MWTF and fault coverage.

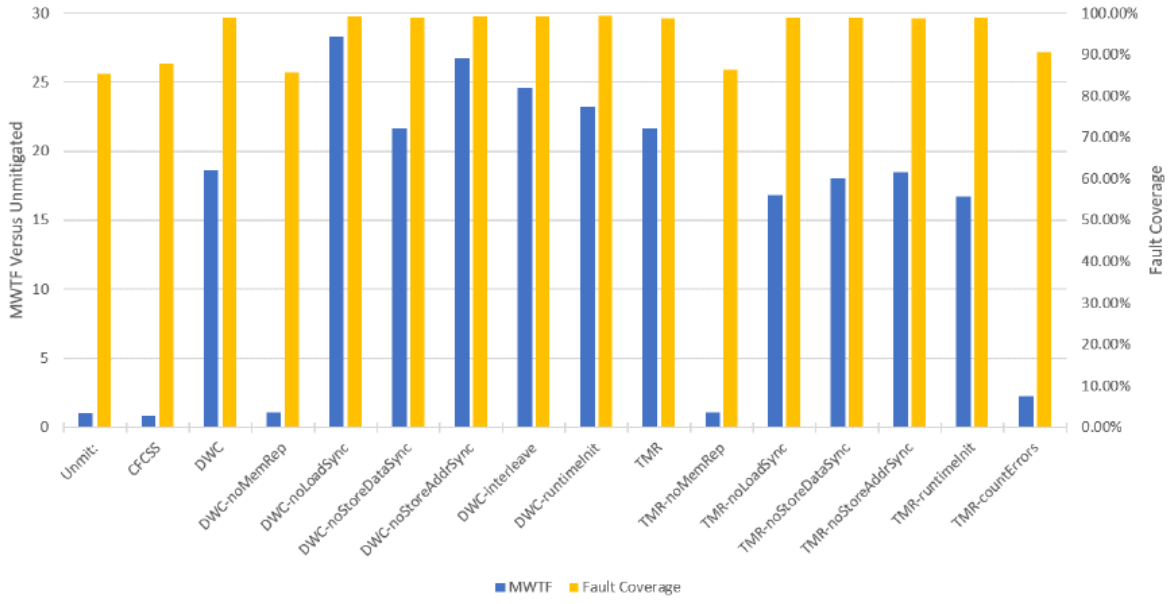
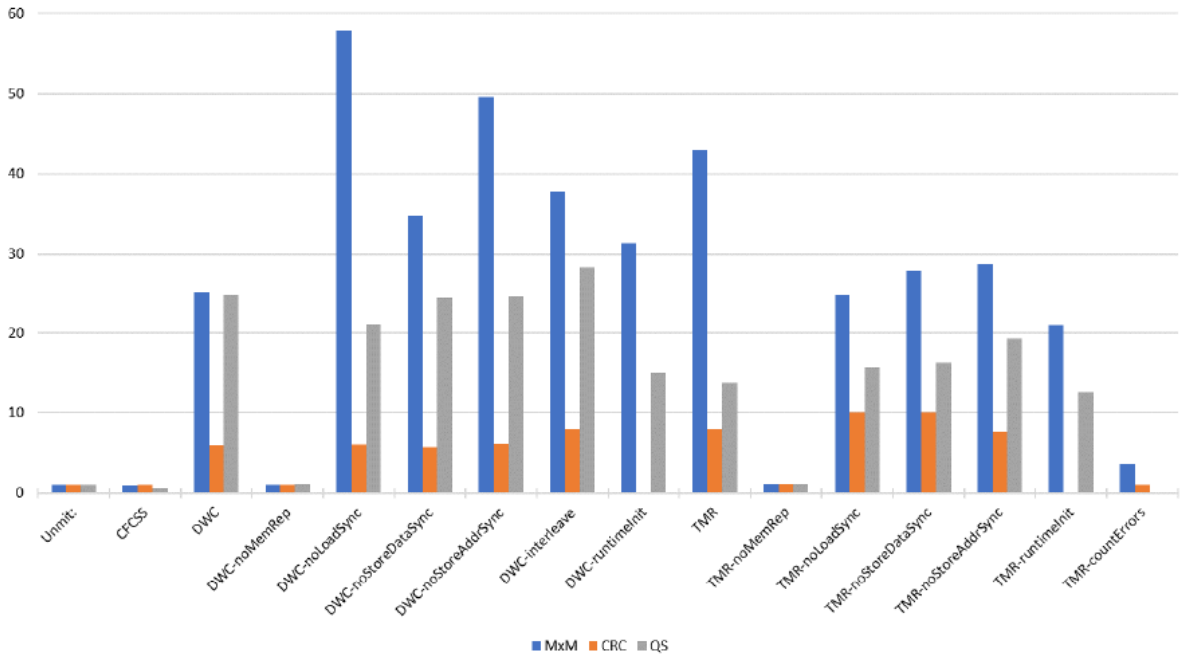


Figure 4.4: MWTF across benchmarks and configurations.



CHAPTER 13

Additional Resources

- Matthew Bohman's Master's thesis.
- IEEE Transactions on Nuclear Science, Vol. 66 Issue 1 - Microcontroller Compiler-Assisted Software Fault Tolerance
- IEEE Transactions on Nuclear Science, Vol. 67 Issue 1 - Applying Compiler-Automated Software Fault Tolerance to Multiple Processor Platforms