

EE/CprE/SE 492 - sdmay21-10

Dynamic Programming Based FPGA Hosted Control-Flow Integrity (CFI)

Client: Akhilesh Tyagi

Faculty Advisors: Zelong Li, Ananda Biswas

Email: sdmay21-10@iastate.edu

Team Members

Gregory Wendt - Meeting Scribe

Cole Schumacher - Meeting Facilitator

Nickolas Mitchell - Chief Engineer (FPGA)

Sam Henley - Chief Engineer (Software)

Maxwell Wangler - Test Engineer

Tristan Duyvejonck - Report Manager

Table Of Contents

1 Revised Project Design	1
1.1 How Our Design Evolved	1
1.2 Functional Requirements	1
1.3 Non-Functional Requirements	1
1.4 Standards	1
1.5 Engineering Constraints	2
1.6 Security Concerns and Countermeasures	2
2 Implementation Details	2
2.1 Race Logic	2
2.2 LLVM	2
2.3 FPGA	3
2.4 Final Design	3
3 Testing Process and Details	3
3.1 Process	3
3.2 Results	4
4 Related Products and Literature	4
4.1 Related Literature	4
Appendix I - Operation Manual	5
I.1 LLVM	5
I.2 Race Logic C Code	5
I.3 FPGA - Altera Quartus Prime	6
Appendix II - Alternative Versions of the Design	8
Appendix III - Other Considerations	8
Appendix IV - Code	9

1 Revised Project Design

1.1 How Our Design Evolved

- Our two input strings (readLabels, readExpected) started as any length ASCII character strings but we changed this in our VHDL code to be eight-bit input strings.
- With a very basic understanding of the algorithm at the beginning of designing, we wanted our race logic to run at $O(n \log(n))$ because we know that was a good run time. Our final design runs at $O(nm)$.

1.2 Functional Requirements

- The algorithm matches two sets of strings (race logic).
 - One string generated based on control flow, and compared to a defined string.
 - Use a 2-D grid of which the size is determined by the number of basic blocks.
- Host machine generates strings based on the sequence of basic blocks using LLVM.
- Host machine and FPGA must communicate with each other.

1.3 Non-Functional Requirements

- The algorithm should run fast. $O(n \log(n))$ is the target runtime.
 - ultimately ended up needing it to be $O(nm)$ in current version

1.4 Standards

- IEEE 1008-1987: IEEE Standard for Software Unit Testing
 - This standard is used to facilitate accurate and consistent unit testing for the software part of the project. This applies as the algorithm the project is built off of must be rigorously tested.
- IEEE 1500-2005: IEEE Standard Testability Method for Embedded Core-based Integrated Circuits
 - This standard is used to facilitate the testing of the hardware side of the project. As the hardware can't be tested in the same methods as the software tests, this standard is applied to the integration between the software and hardware.
- IEEE 15288-2004: Systems and Software Engineering System Life Cycle Processes
 - This standard is used in the design and development of a framework in which the software can maintain a healthy life cycle. This included a section about how this deals with . This applies to the project as it helped guide the development so the project is stable after development.
- IEEE 1220-2005: IEEE Standard for Application and Management of the Systems Engineering Process
 - This standard is used to create necessary infrastructure for life cycle sustainment. This applies to the project as it helps build out that infrastructure necessary for post development implementation of the project.

1.5 Engineering Constraints

- We need to use dynamic programming to determine if CFI was lost.
- We need to use an FPGA board to test the solution.
- The total project cost should not exceed \$100.
- The project needs to be completed by the end of CprE 492.

1.6 Security Concerns and Countermeasures

An adversary could gain access to our physical hardware or to our program through software. In both of these cases, an adversary could tamper with the control flow graph and change what is passed to our algorithm. To fix the hardware security concern, we propose keeping the FPGA in a well protected section of a building. To fix the software security concern, we propose keeping a log on where information comes from. This way we will be able to manually identify manipulation in race logic.

2 Implementation Details

2.1 Race Logic

The race logic algorithm has gone through many iterations. Starting as a pseudocode algorithm, it was then implemented in java and tested using JUnit 5 testing framework. Then it was implemented in C and tested with a custom testing framework developed for this project. Finally, it was translated into verilog. The algorithm works by taking in two character arrays sized n and m respectively. Then, it creates an integer matrix size n by m that then has all its entries initialized to 0. Once that's complete the system goes into a nested for loop that iterates over each element in the array starting with $[0,0]$ and ending with $[n-1,m-1]$. At each step, the algorithm checks if the two arrays equal one another at the current position in the matrix, i. e. if the first loop is at step 2 and the second is at step 4, the algorithm will check if $array1[2]$ equals $array2[4]$. If they are equal, the array will then set the location one step down and to the right of it to its current value plus one, if it is able and if the new location does not have a non-zero value less than the new value. Then, the algorithm will place it's value plus one in the location one step down and the location one step to the right if they don't have a non-zero value less than the new value. Once that is done for each element in the matrix, the system will return the element at $[n-1,m-1]$ as the score for these two arrays.

2.2 LLVM

In order to get the necessary data from the program, we are using a software called LLVM. LLVM is an open-source compiler that converts source code into a language called Intermediate Representation before a system-specific binary is generated. Development regarding LLVM was performed on an Ubuntu Virtual Machine. After installing LLVM, we developed two "passes", which are optimizations or transformations LLVM performs on the Intermediate Representation.

- 1) BlockLabelsPass: This pass iterates over each Basic Block in the program. When the pass encounters a Branch instruction, it outputs a unique identifier associated with the current Basic Block. This pass is used to generate the static “expected” sequence of Basic Blocks encountered before the program is executed.
- 2) InjectFunctionPass: This pass iterates over each Basic Block in the program. When the pass encounters a Branch instruction, it inserts a call to a separate function, passing the unique identifier associated with the current Basic Block as a parameter. This separate function is used to output the current Basic Block when the program is executed. This pass is used to generate the dynamic “actual” sequence of Basic Blocks encountered during execution.

2.3 FPGA

Development regarding FPGA was performed in Quartus Prime and then uploaded over USB to the Altera FPGA board. Using Quartus Prime our Java and C race logic codes were translated into VHDL code. With our race logic code (described in section 2.1) running on a FPGA board, a computer will be sending the data our algorithm needs over UART on a USB cable. This communication link is described in more detail in section 2.4 (Final Design).

2.4 Final Design

Our final design closely resembles figure 2.1 below. Given our Altera FPGA board and a host computer, we have LLVM (described in section 2.2) running on the computer to extract the binary from a running program which is then sent to the FPGA board (described in section 2.3). The FPGA then reads these two labels into the race logic algorithm (described in section 2.1) and then sends the result back to the computer.

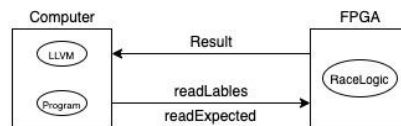


Figure 2.2: Overall Design

3 Testing Process and Details

3.1 Process

Race Logic Testing:

To test the race logic, we constructed a suite of unit tests to test both primary and edge cases of this system. To do this, we took the algorithm and isolated it with controlled input strings to determine that the faults were coming from the algorithm itself. To guarantee coverage, we then build tests for each combination of strings that could be entered.

LLVM Testing:

Testing the LLVM passes was performed by writing several sample programs to invoke the passes on. We then compared the outputs of each pass before sending them to the race logic algorithm.

3.2 Results

Race Logic:

The results of our race logic tests are 13 passing tests in JUnit or a Custom C test framework that verify multiple edge cases of the algorithm work.

LLVM:

Towards the end of our development window, we discovered that the passes were not giving the result we expected. BlockLabelsPass was expected to follow the control flow of the program and print the sequence of basic blocks that should be encountered. However, the pass simply iterates over a list of basic blocks contained in the program, not necessarily following the control flow path. Similarly, the InjectFunctionPass was not providing the desired output.

4 Related Products and Literature

4.1 Related Literature

A. Madhavan, T. Sherwood and D. Strukov, "Race Logic: Abusing Hardware Race Conditions to Perform Useful Computation," in *IEEE Micro*, vol. 35, no. 3, pp. 48-57, May-June 2015, doi: 10.1109/MM.2015.43.

Gomez-Martinez, G. (n.d.). UART TRANSMISSION ON THE DE2-115 BOARD. Retrieved April 1, 2021, from <http://cmosedu.com/jbaker/students/gerardo/Documents/UARTonFPGA.pdf>

The *Race Logic: Abusing Hardware Race Conditions to Perform Useful Computation* paper by Madhavan, Sherwood, and Strukov was useful in understanding the algorithm we use in the system. The authors applied race conditions to solve a string-matching problem related to protein sequencing. We adapted this algorithm to analyze control flow integrity. The *UART TRANSMISSION ON THE DE2-115 BOARD* article by Gomez-Martinez goes over the details of communication between a FPGA board and computer over UART. This idea was used to guide us in how we set up communication between our FPGA board and computer.

Appendix I - Operation Manual

I.1 LLVM

Development and testing for the LLVM module was performed on a virtual machine running Ubuntu. To invoke LLVM passes on a target program (i.e. target.c):

1. Install LLVM

```
sdmay21-10/llvm$ sudo apt-get install llvm
```

2. Build the BlockLabelsPass

```
sdmay21-10/llvm$ cd BlockLabelsPass
sdmay21-10/llvm/BlockLabelsPass$ mkdir build
sdmay21-10/llvm/BlockLabelsPass$ cd build
sdmay21-10/llvm/BlockLabelsPass/build$ cmake ..
sdmay21-10/llvm/BlockLabelsPass/build$ make
```

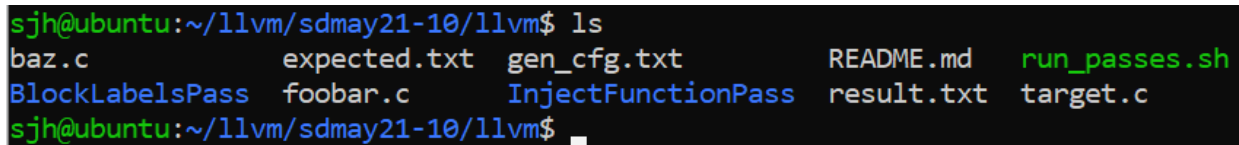
3. Build the InjectFunctionPass

```
sdmay21-10/llvm/BlockLabelsPass/build$ cd ../../InjectFunctionPass
sdmay21-10/llvm/InjectFunctionPass$ mkdir build
sdmay21-10/llvm/InjectFunctionPass$ cd build
sdmay21-10/llvm/InjectFunctionPass/build$ cmake ..
sdmay21-10/llvm/InjectFunctionPass/build$ make
```

4. Run the shell script to invoke both passes on the target source file

```
sdmay21-10/llvm/InjectFunctionPass/build$ cd ../../
sdmay21-10/llvm$ ./run_passes.sh target.c
```

The outputs of BlockLabelsPass and InjectFunctionPass are stored in “expected.txt” and “result.txt”, respectively.



```
sjh@ubuntu:~/llvm/sdmay21-10/llvm$ ls
baz.c          expected.txt  gen_cfg.txt   README.md    run_passes.sh
BlockLabelsPass foobar.c     InjectFunctionPass result.txt   target.c
sjh@ubuntu:~/llvm/sdmay21-10/llvm$
```

Figure I.1: location of “expected.txt” and “result.txt” files after executing run_passes.sh

I.2 Race Logic C Code

Development and testing for the Race Logic code was performed on a Window 10 machine. To run the test suite for the C code:

1. Navigate to the C_Code directory.

```
sdmay21-10$ cd C_Code
```

2. Make the necessary files by running the make command.

```
sdmay21-10/C_Code$ make
```

3. Run the TestSuite Executable File.

```
sdmay21-10/C_Code$ ./TestSuite
```

This should print out to the terminal what tests succeeded and what tests failed out of the thirteen tests implemented in the test suite.

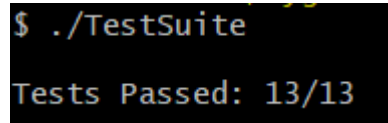


Figure I.2: Output of the TestSuite code

To run the race logic algorithm, include RaceLogic.h as a header in the C file. The function is `init_RaceLogic(unsigned char* acc1, unsigned char* ex1)`. Where `acc1` and `ex1` are both arrays of unsigned chars.

I.3 FPGA - Altera Quartus Prime

Please note, this section goes over how to flash any particular code onto the Altera DE2-115 FPGA board. Using Altera Quartus Prime, either verilog or VHDL code can be written then flashed onto the FPGA board.

1. Open the program in Altera Quartus Prime and make sure the correct model FPGA board is selected. To select the correct FPGA board, double click the box highlighted in figure I.3 and then select the correct board ID from the list. Board ID: EP4CE115F29C7.
2. After selecting the correct FPGA board, compile the code by clicking the plain blue triangle next to the greyed out stop button. After compiling the code it is ready to be flashed to the FPGA board.
3. Open the programmer which is highlighted in figure I.4.

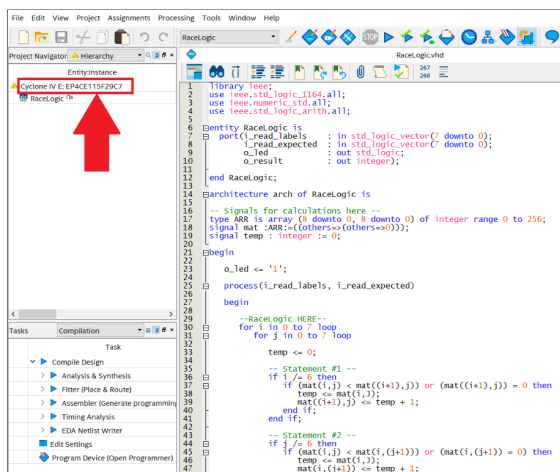


Figure I.3

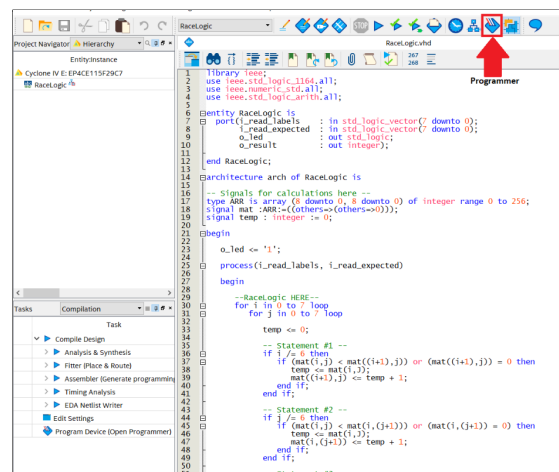


Figure I.4

4. After opening the programmer a window will popup as depicted in figure I.5. Inside of this window select hardware setup and inside of that window select the no hardware drop down and select the USB-Blaster. Your screen should now look like figure I.6.

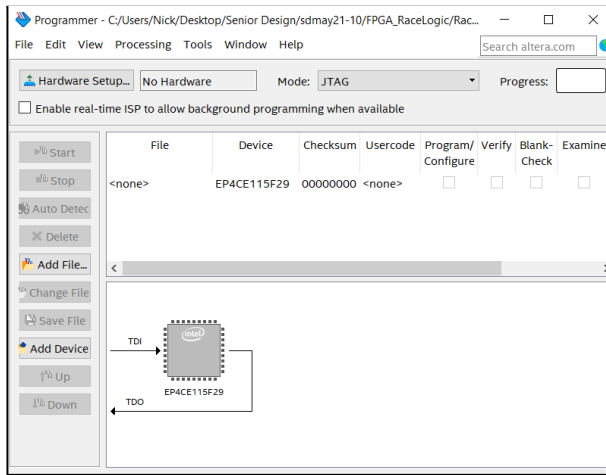


Figure I.5

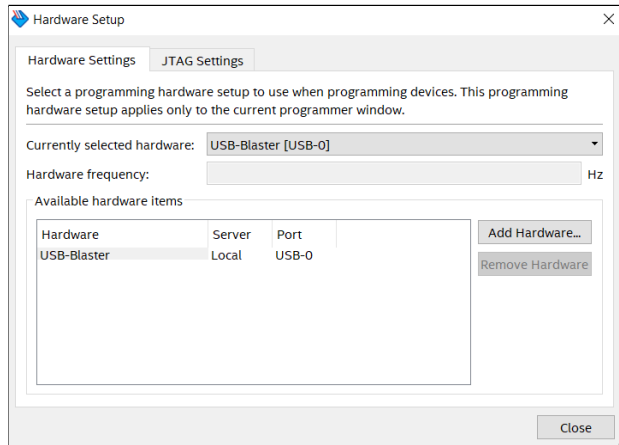


Figure I.6

5. Lastly, click start in the programmer window. Congratulations, your code has now been flashed to the Altera DE2-115 FPGA board and is currently running. To stop your code just click the stop button.

Common errors when flashing your code to the DE2-115 FPGA board.

1. The device does not show up in the programmer window.
 - a. Make sure your FPGA board is connected to the correct USB port on the board and the board is turned on. The FPGA board will have many cool lights if it is turned on.
 - b. Make sure the FPGA drivers are up to date. The most up to date drivers can be found here: <https://fpgasoftware.intel.com/?edition=lite>
2. The programmer window does not allow you to select start.
 - a. Make sure your program has successfully compiled with no errors.

Appendix II - Alternative Versions of the Design

Previous versions of race logic include the Java and C code we created in computer engineering 491. A link to this code can be found in Appendix IV. Our VHDL solution is the most up to date version of our design for the race logic code.

Appendix III - Other Considerations

- This project was worked on entirely over video chat. We had no in person meetings due to the COVID-19 pandemic.
- Our group had a minimal understanding of Control Flow Integrity, dynamic programming, and FPGA logic at the beginning of this project. Many of our early meetings consisted of discussions regarding the meanings of each of these concepts and how we could combine them for this project.

What went well:

- Designing and testing the race logic algorithm as a team.
- Consistent communication with client
- Dividing project into manageable sections (LLVM, Race Logic, FPGA)

What could have gone better:

- LLVM
 - A better understanding or experience using LLVM would have been very beneficial going into this project. The documentation for LLVM is very complex, and a large portion of time at the start of our development window was spent working towards becoming more proficient with LLVM.
- FPGA
 - On the FPGA side of things, we completed translating the C and Java code to VHDL but didn't get the opportunity to test the VHDL code due to not finishing the communication link. Our final design relied on a communication link between the FPGA and computer using UART over USB. This proved to be very challenging and after many hours as well as working with our two graduate students (Ananda and Zelong), we did not complete this part of the project.
 - In an effort to spend more time working on the communication link, our team chose to only use Quartus Prime to develop the FPGA solution. Because of this, Quartus Prime has no built in simulator to the best of our knowledge, therefore we were not able to test the VHDL solution. Moreover, we were confident with the testing we did on the C and Java code therefore, felt it was not necessary to test the FPGA code. Had we not made the decision to spend more time working on the communication link, we planned to test our solution using Modelsim which does have a built in simulator.

Appendix IV - Code

Please note, all code for our project can be found [here](#).

<https://git.ece.iastate.edu/sd/sdmay21-10>

VHDL Race Logic:

https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/master/FPGA_RaceLogic/RaceLogic.vhd

Java Race Logic:

[https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/master/Dynamicrace logic/src/race logic.java](https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/master/Dynamicrace%20logic/src/race%20logic.java)

C Race Logic:

[https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/master/C_code/race logic.c](https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/master/C_code/race%20logic.c)

BlockLabels (LLVM):

<https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/llvm/llvm/BlockLabelsPass/blockLabels/BlockLabels.cpp>

Inject Function (LLVM):

<https://git.ece.iastate.edu/sd/sdmay21-10/-/blob/llvm/llvm/InjectFunctionPass/injectFunc/InjectFunc.cpp>