Dynamic Programming Based FPGA Hosted Control-Flow Integrity

Group: sdmay21-10
Gregory Wendt, Cole Schumacher, Nickolas Mitchell, Sam Henley, Maxwell Wangler, Tristan Duyvejonck

Client: Dr. Akhilesh Tyagi **Advisors: Zelong Li, Ananda Biswas**

Introduction/Motivation

Control-flow integrity (CFI) is an important attribute of a program execution in cybersecurity engineering. CFI means that during the program execution, a control flow edge that is not part of the program's control flow graph (CFG) should not be taken. This project entails developing a dynamic programming based model of CFI based on some CFG attributes. This project and an additional model with FPGA 2-D array solver using delay stacking that are to be combined into a control flow integrity engine.

Design Requirements

Functional:

- The algorithm matches two sets of strings (race logic).
  - One string 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.

Non-Functional:

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

Technical Details

Race Logic Algorithm:

- 8-bit Inputs
  - The two input strings are 8-bits each.
  - The result is an integer between 0-255 depending on how different the two strings are.
  - The larger the result, the more likely control flow has been broken

LLVM:

- Get the sequences of basic blocks
- Develop passes - transformations to program before compilation

FPGA:

- VHDL
  - Using Quartus Prime to Implement
- UART communication between computer and FPGA
  - FPGA receives two strings from computer to run through our VHDL race logic then the result is returned to the computer.

Intended Users and Uses

- Intended users of this project include IT personnel and any company concerned with cyber security.
- Intended uses include the ability to detect faults in race logic and security for networks.

Design Approach

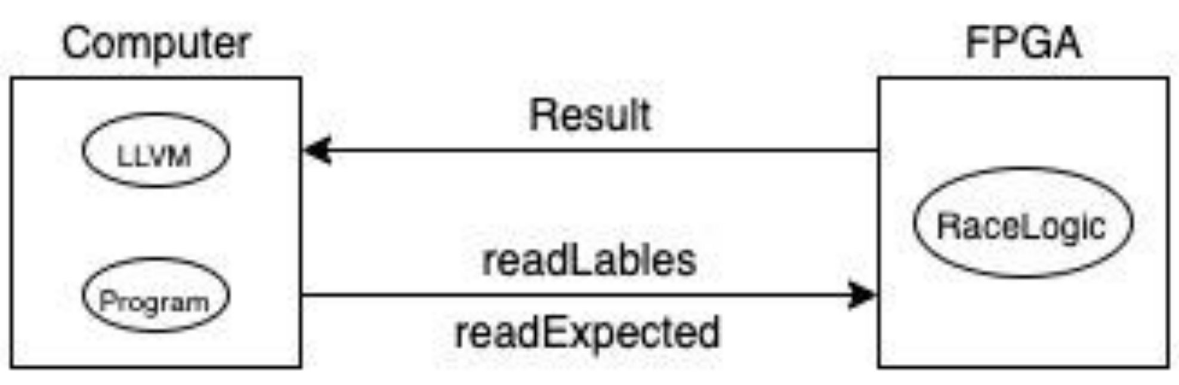

Figure 1: Overall Design

The figure above depicts our overall design. When approaching this design, we decided to break our project down into different modules. The different modules include RaceLogic, LLVM, and FPGA. Connecting these three modules gives us our overall design.

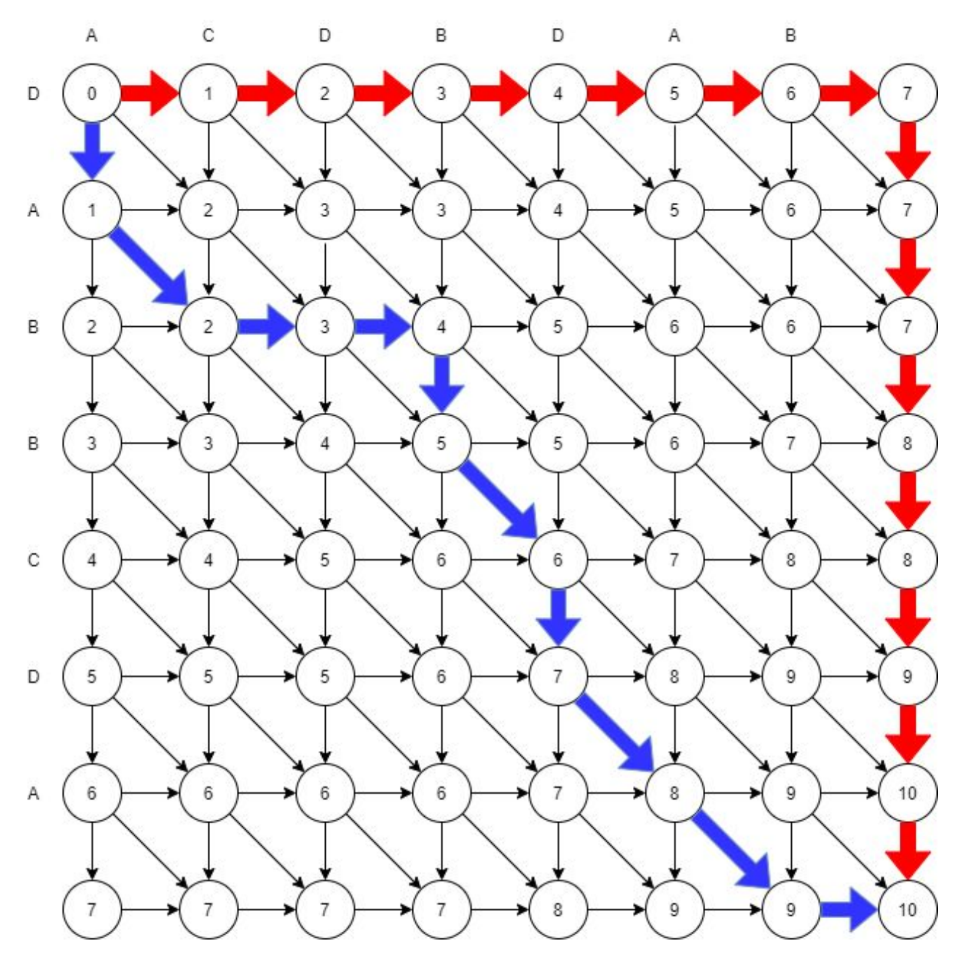

Figure 2: Race Logic

The figure above is an example of the logic behind our Race Logic Algorithm. The Blue path is the most efficient way to travel from the upper left hand corner to the lower right hand corner using only legal moves. The Red is the most inefficient path.

Testing

JUnit:

- In early versions of our racelogic, we used JUnit to see if our algorithm works. When we moved to C, these same tests were recreated with C compatible testing methods.

C test suite:

- A custom C test framework was developed to test our C code when it was migrated. This was created to verify that our algorithm could still function.

LLVM:

- Several sample programs were developed and sent through the LLVM passes, then we compared the results of each pass.

Standards Used

IEEE 1008-1987

IEEE 15288-2004

IEEE 1500-2005

IEEE 1220-2005