Presentation Outline

1. Intro
   1. My programs
      1. Verifier\_Round\_3
      2. Random Program Generator
   2. What problem is it attempting to solve
      1. Taking a valid eBPF program, run it through a FOL translator to see if the output matches what eBPF Verifer says it should be. If there is a mismatch, it can indicate a bug in the eBPF Verifier, or a bug in the FOL Verifier
      2. How do you automate program generation to test a wide range of different programs to see the depths of both the eBPF Verifier and the FOL Verifier
   3. What are its limitations
      1. FOL Verifier
         1. Small subset of eBPF instructions supported
      2. Random Program Generation
         1. Unknown chance to produce viable programs for testing due to current random nature
2. Specifics about Verifier\_Round\_3
   1. Internal Logic for Naming Scheme
      1. SSA Naming for every instance a register has a value changed
      2. Required for z3
      3. Initialized by create\_register\_list, which creates a 2D List containing placeholders for the number of registers allowed in the program
      4. Updated by appending a new Register\_Info object to the end of a specific registers sublist
      5. Stored in register\_state\_helper.register\_history
   2. Valid input checking and extension
      1. Any time a function is called, there are three options for getting a value
      2. Imm4, imm8, register
      3. Imm4 would be using a value that, bit wise, is half the size of the regular register
      4. Need to extend the size of the value to not have sign problems
      5. Use extend\_the\_number function to encapsulate both the sign extension, or zero extension needs
   3. Supported eBPF Functions
      1. All functions are generic calls, which can be made to do specific (imm4, imm8, or reg) versions of the call by using the source\_reg and extension\_length parameters
         1. Source\_reg is a Boolean stating whether the input\_value parameter should be treated as a register number or an outside integer
         2. Extension\_length is used to tell extend\_the\_number how many bits a specific imm4 number needs to be lengthened to fit in the register
      2. Add
         1. Add two values and create a new instance of the destination register for ssa
         2. Prevent overflow
         3. Prevent underflow
      3. Mov
         1. Sets a new instance of the destination register equal to whatever the incoming value is
      4. Jne
         1. Get the names (ie reference points) for all registers before the jump instruction
         2. Execute the code that would be taken without the jump
         3. Get the names of all registers after the non jump execution
         4. Set a new layer of registers across all initialized registers using an if, then, else z3 function
         5. Return a skip flag that is larger than the instruction counter to tell the main program to skip all instructions until the offset, since this will return a z3 function containing their constraints to the main program, and we don’t want to double up on instructions
   4. Execution of a specific program
      1. Create\_program
         1. User seen program call
         2. Parameters
            1. Program\_list

Default is blank, will call a built in program as a backup

* + - * 1. Num\_regs

Default is 4

* + - * 1. Reg\_bit\_width

Default is 8

* + - 1. Creates the register\_state\_helper, keeps track of program execution time, and calls execute\_program
    1. Execute\_program
       1. Main Workhorse
       2. Iterates through the register\_state\_helper.instruction\_list
       3. Executes each instruction by passing it into create\_new\_constraints
          1. C\_n\_c returns a z3 boolean equation and an updated r\_s\_h
       4. Combines the returned z3 Boolean equation with the current solver, and checks if function is satisfiable
          1. If sat, moves onto the next instruction
          2. If unsat

Sets r\_s\_h.problem\_flag to tell which instruction caused the problem

Reverts the solver back to a state before the problematic instruction

* + - 1. If function iterates through entire instruction list, or hits a problem\_flag
         1. Prints out the contained z3 solver
         2. Prints out the end results for all distinct registers
         3. Prints out a string containing a translated version of the Python instruction list as an eBPF macro list for use in bpf\_step
    1. Create\_new\_constraints\_based\_on\_instruction
       1. Splits an individual instruction into keyword, source\_reg or value, target\_reg, and offset value (for jump commands)
       2. Every class of instructions (add, mov, or jne) has three versions, depending on how the first number in the keyword is interpreted.
          1. The final two params of add\_two\_values, mov\_to\_reg, and jump\_general are used to differentiate between imm4, imm8, and register source values
       3. Every instruction returns the same two things, a new group of z3 constaints, and an updated register\_state\_helper with new register names and possible problem\_flag updates

1. Specifics about Random Program Generator