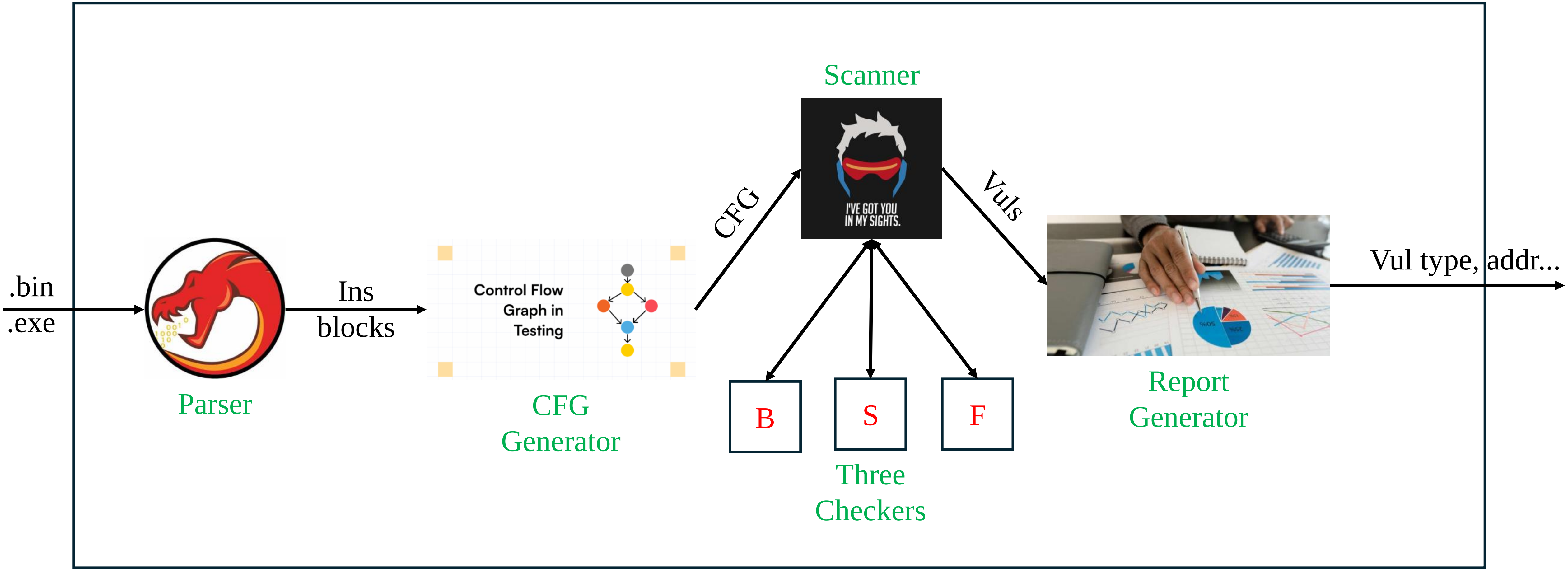
32-bit x86 Binary Vulnerability Checker

Boning Li & Lianting Wang

Project Structure

```
x86-vulnerability-checker/
│
├── app/
│   ├── Main.hs                  # Entry point for the executable
│
├── src/
│   ├── Types.hs                 # defines data structure for vulnerability
│   ├── BinaryParser.hs          # Parses binaries and extracts instructions
│   ├── CFGGenerator.hs          # Generates CFG from instructions
│   ├── CFGPrinter.hs            # Prints CFG in a human-readable format
│   ├── VulnerabilityScanner.hs # Scans CFG for vulnerabilities
│   ├── FormatStringChecker.hs  # Detects format string vulnerabilities
│   ├── StackIntegrityChecker.hs  # Detects stack integrity vulnerabilities
│   ├── BufferOverflowChecker.hs  # Detects buffer overflow vulnerabilities
│   └── ReportGenerator.hs       # Generates vulnerability report
│
├── test/
│   ├── <module>Spec.hs          # Unit test for a module
│   └── Spec.hs                  # Basic test cases for each module
│
├── x86-vulnerability-checker.cabal # Project configuration and dependencies
├── README.md                    # Project README
└── LICENSE                      # License information
```

Workflow


.bin
.exe
Ins
blocks
Parser
Control Flow
Graph in
Testing
CFG
Generator
CFG
Scanner
I'VE GOT YOU
IN MY SIGHTS.
Vuls
B
S
F
Three
Checkers
Report
Generator
Vul type, addr...

Unit Tests

- BinaryParserSpec.hs
- BufferOverflowCheckerSpec.hs
- CFGGeneratorSpec.hs
- FormatStringCheckerSpec.hs
- Spec.hs
- StackIntegrityCheckerSpec.hs
- VulnerabilityScannerSpec.hs

```haskell
main = hspec $ do
  describe "Binary Parsing Tests" BinaryParserSpec.spec
  describe "CFG Generator Tests" CFGGeneratorSpec.spec
  describe "Format String Checker Tests" FormatStringCheckerSpec.spec
  describe "Buffer Overflow Checker  Tests" BufferOverflowCheckerSpec.spec
  describe "Stack Integrity Checker Tests" StackIntegrityCheckerSpec.spec
  describe "Vulnerability Scanner Tests" VulnerabilityScannerSpec.spec
```

```haskell
it "generates a CFG with a single basic block for linear code" $ do
  let instructions = [ Instruction "1000" "55"     "push" "ebp"
                     , Instruction "1001" "89 e5"  "mov"  "ebp, esp"
                     , Instruction "1003" "5d"     "pop"  "ebp"
                     , Instruction "1004" "c3"     "ret"  ""
                     ]
  let cfg = generateCFG instructions
  -- Check that there is one basic block
  Map.size (cfgBlocks cfg) `shouldBe` 1
  -- Check that there are no edges
  let totalEdges = sum $ Prelude.map length $ Map.elems (cfgEdges cfg)
  totalEdges `shouldBe` 0
```

Best Module

- My favorite module is CFGGenerator. It is the most important module in our project and every vulnerability checker works based on it. Besides, it is also the part I'm most familiar with.

```haskell
sortedAddrList = sort $ Map.keys addrToBBLabel

-- Determine the successors of a given basic block
getSuccessors :: BasicBlock -> [String]
getSuccessors bb =
  if null (bbInstructions bb)
    then []
    else
      let lastInstr = last (bbInstructions bb)
          mnem      = mnemonic lastInstr
      in case mnem of
           "jmp" -> getJumpTargets lastInstr
           m | m `elem` ["je", "jne", "jg", "jge", "jl", "jle", "ja", "jb"] ->
                 let jumpTargets = getJumpTargets lastInstr
                     fallThrough = getFallThrough (parseAddress (address lastInstr))
                 in jumpTargets ++ fallThrough
           "ret" -> []
           "call" ->
             let callTargets = getCallTarget lastInstr
                 fallThrough = getFallThrough (parseAddress (address lastInstr))
```

```haskell
-- Function to parse hexadecimal addresses
parseAddress :: String -> Integer
parseAddress s = case readHex s of
                   [(addr, "")] -> addr
                   _            -> error $ "Invalid address: " ++ s

-- Collect starting addresses of basic blocks
collectBasicBlockStarts :: [(Instruction, Integer)] -> Set Integer
collectBasicBlockStarts instrAddrs = Set.fromList $ startAddr : jumpTar
  where
    startAddr = snd (head instrAddrs)
    jumpTargets = mapMaybe getJumpTarget instrAddrs
    postJumpAddrs = mapMaybe getPostJumpAddress (zip instrAddrs (tail i

    -- Extract jump and call targets from instructions
    getJumpTarget (instr, _) =
      if mnemonic instr `elem` jumpMnemonics
        then parseOperandAddress (operands instr)
        else Nothing
      where
        jumpMnemonics = ["jmp", "je", "jne", "jg", "jge", "jl", "jle",

    -- Extract post-jump addresses (instructions immediately following
    getPostJumpAddress ((instr, _), (_, nextAddr)) =
      if mnemonic instr `elem` blockEndMnemonics
        then Just nextAddr
        else Nothing
      where
        blockEndMnemonics = ["jmp", "je", "jne", "jg", "jge", "jl", "jl
```

Future Improvement

Improve the parser so that it’s able to recognize more mnemonics.

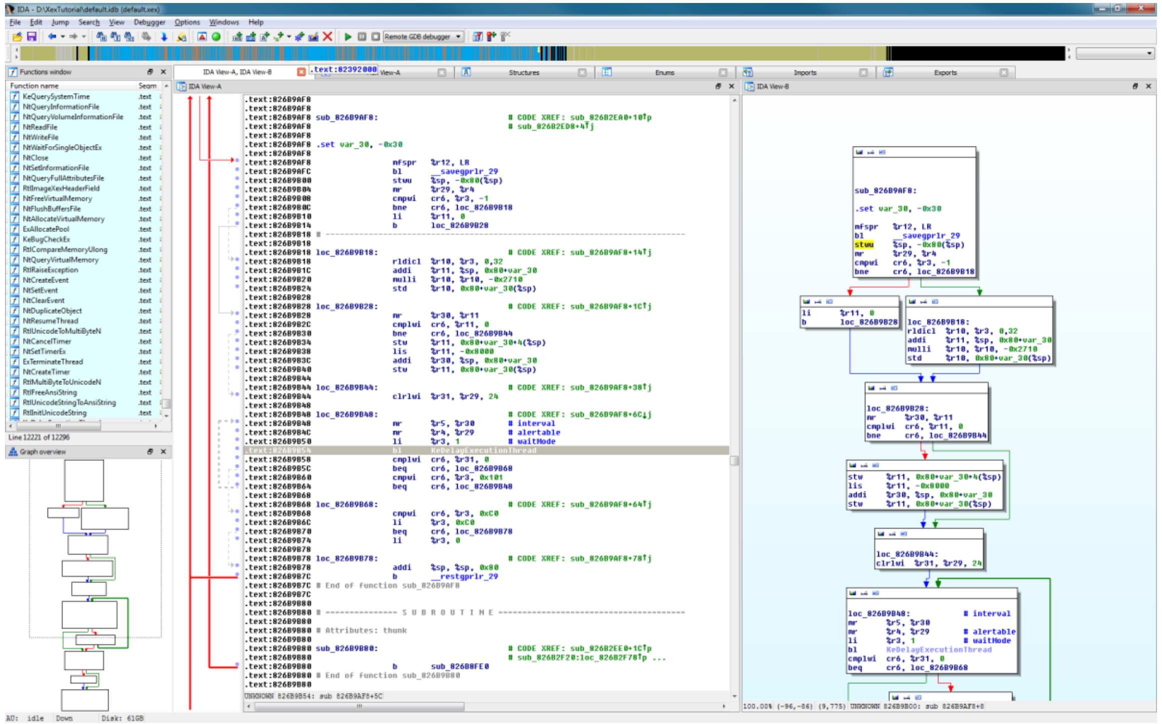
Extend CFG so that it works with program with multiple and external function calls, and other essential functionalities.

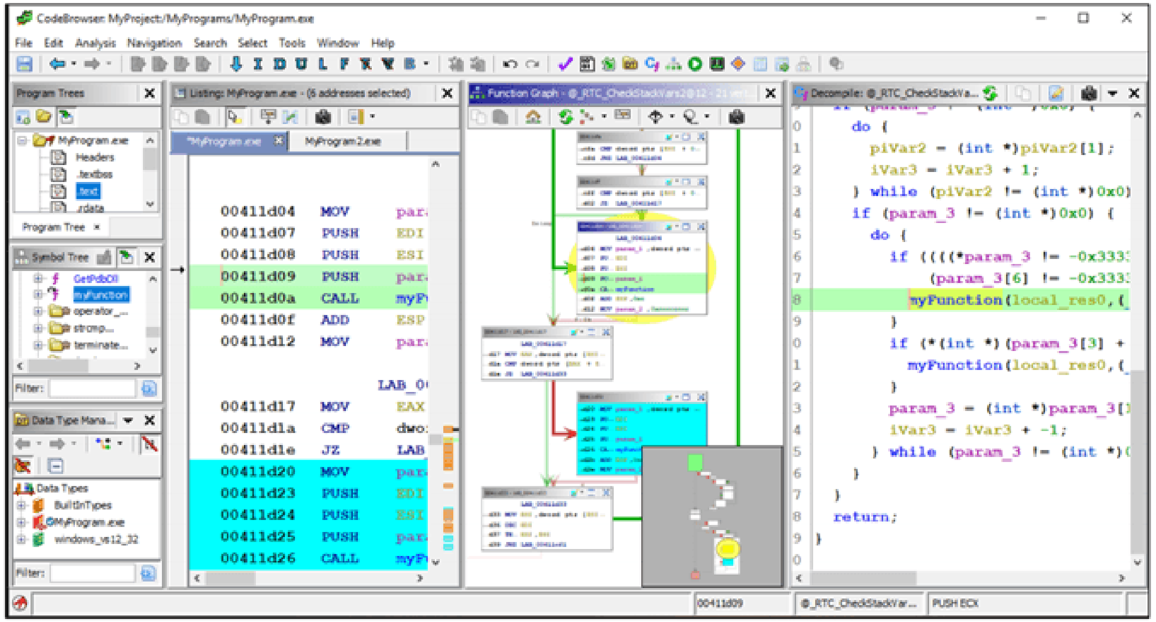
Refine the definition of each vulnerability to make checkers work better.

Design a better visualization of output

Implementation in other languages

- There are more powerful tools for reverse engineering and static binary analysis, such as Ghidra and IDA. For example, Ghidra provides API for Java and Python. It will be much easier to implement all functionalities and to do much better using Ghidra.





Summary

- Through this project, we
  - Reviewed the concepts of static control flow graph
  - Learnt how to detect vulnerabilities through CFG, e.g. checking stack balance when there are conditional jumps
  - Learnt how to write an effective program