# Source code analysis

## Summary

In this class exercise, we will introduce you to source code analysis.

## Prerequisites

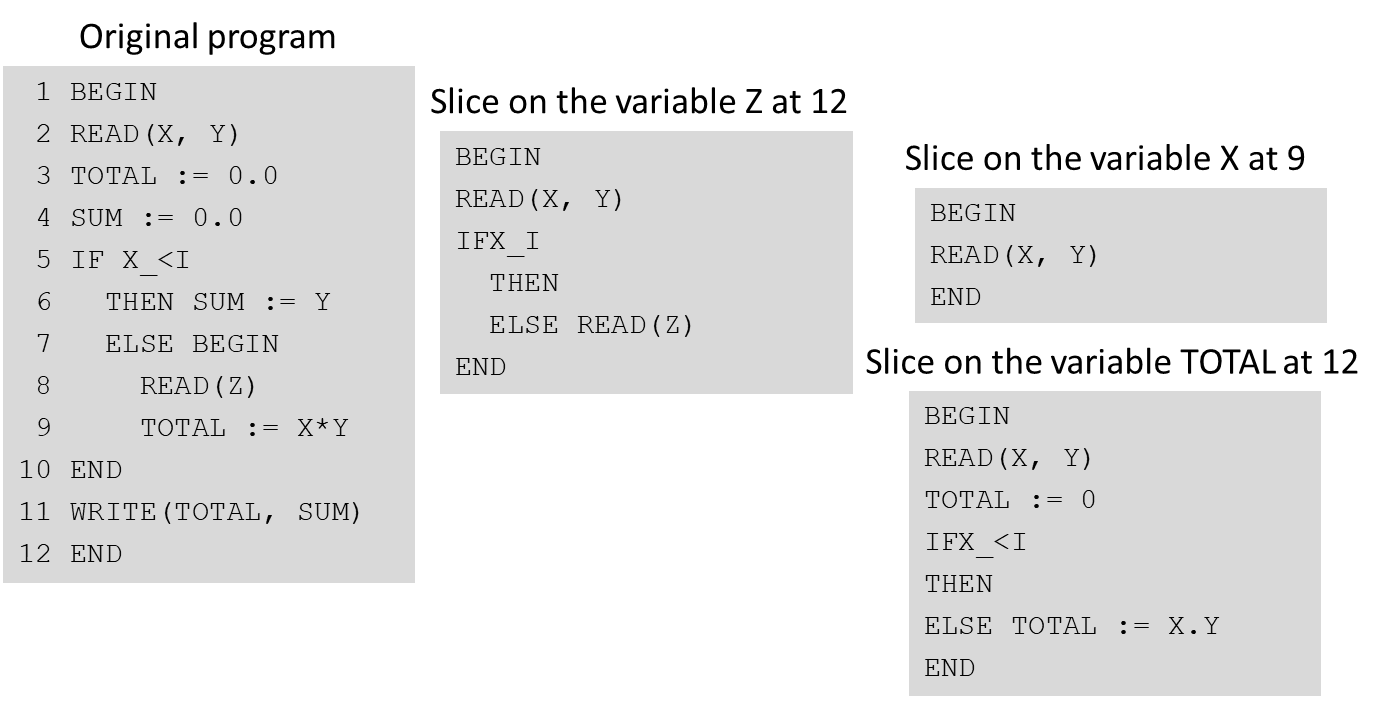
* Ubuntu 18.04 LTS virtual machine and SciTools Understand

## Details

Source-code analysis alternatives

Perform manual analysis

* Talking to developers
* Reading the documentation and results from tools below
* Slicing through the source code



Using semi-automated analysis tools for

* Performing security audits (when technologies support it)
  + E.g., see Lynis (<https://cisofy.com/solutions/#lynis>)
* Providing visualizations
  + Control flow graphs, data flow graphs, dependency graphs, call graphs
* Performing other comprehension or documentation aids
  + Annotation
  + Cross-referencing
* Generating metrics
  + Size, complexity, and object oriented metrics (e.g., cohesion, coupling)

### Part I

Code review sequence

* Set clear objectives for the review
  + A focused review is more effective
  + Spend time at the beginning to understand which security issues are possible
* Set a time limit for the review
  + Time bounding helps focus the review
  + Optimize the review for the time limit by using time management to prioritize
* Utilize question lists that are applicable to the implementation under review
  + Includes technologies and design
  + Questions should support mitigating known vulnerabilities that apply to the technology and design
* Review iteratively
  + Scope bounding helps focus the review
  + Limit the review iterations to small, manageable pieces of code
* Review only for security
  + Code quality issues should be handled separately
* Know your application architecture
  + Component architecture
  + Dataflow from the user, between components, and to any data repositories
* Update your coding standards with lessons learned
  + Code standards
  + Security question lists

Review techniques

* Perform preliminary scan
  + Run automated tools
  + Manually scan for common issues in areas
    - Performing input data validation
    - Authenticating and authorizing users
    - Handling errors
    - Having complex code
    - Performing cryptography
    - Where trust boundaries or privilege levels are crossed
* Perform control flow analysis, a mechanism used to step through logical conditions in the code

1. Examine a function and determine each branch condition (includes loops, switch statements, if statements, and try/catch blocks)
2. Understand the conditions under which each block will execute.
3. Move to the next function and repeat

* Perform data flow analysis, a mechanism used to trace data from the points of input to the points of output
  + Because there can be many data flows in an application, use your code review objectives and the flagged areas from the preliminary scan for focus
  + Prioritize where trust boundaries are crossed

1. For each input location, determine how much you trust the source of input (default is untrusted)
2. Trace the flow of data to each possible output (note any data validation areas)
3. Move to the next input and continue

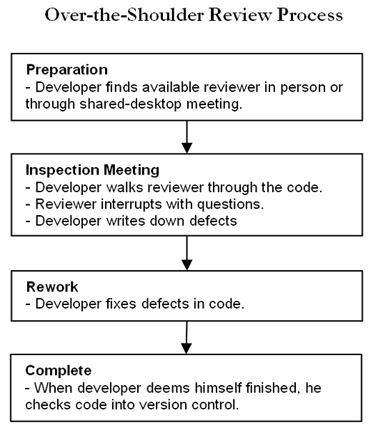


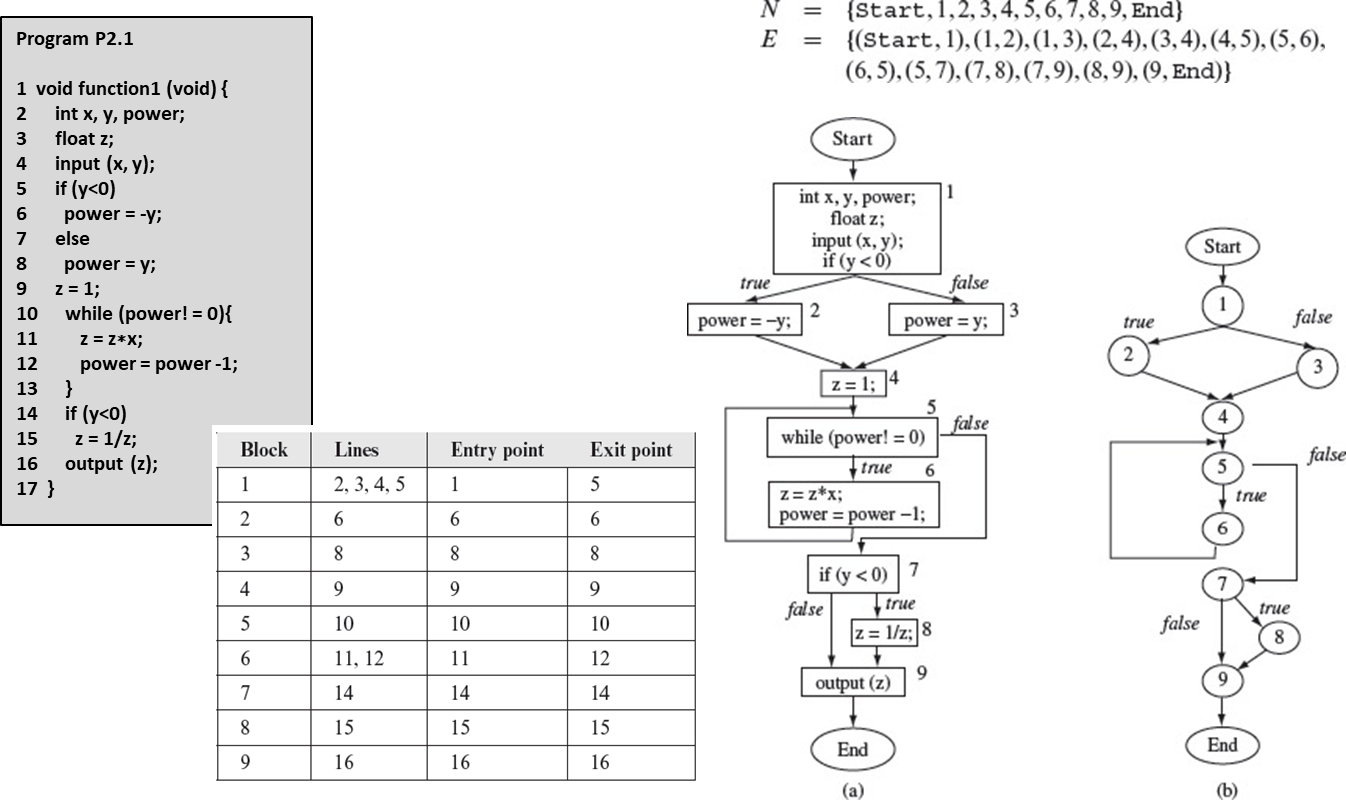
Figure -http://www.methodsandtools.com/archive/archive.php?id=66

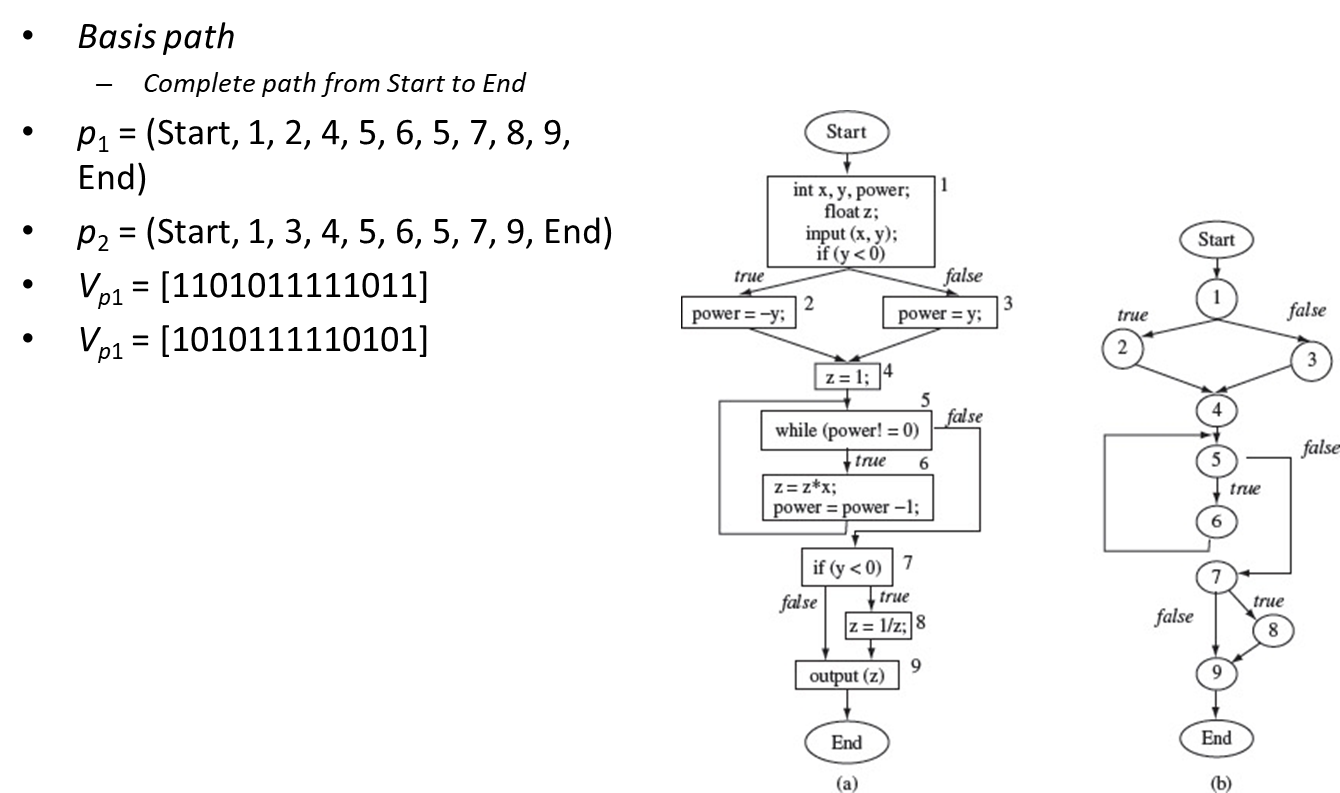
Views

* Control flow graphs
* Dependency graphs
* Declaration graph
* Cluster call graphs

Control flow graph

* Control flow graph
  + Flow graphs (G) consist of a finite set of nodes (N) and directed edges (E)
    - Each node represents a basic block in the program
    - Each edge represents the flow of control from its source node to the destination node
* Basic block
  + Longest possible sequence of consecutive statements in a program such that control can enter the block only at the first statement and exit from the last
  + There cannot be any conditional statement inside a basic block other than at its end
* Path
  + Sequence of edges in the flow graph that indicates the flow of control in the corresponding program
  + Not all paths extracted from a flow graph might be feasible
    - Implies that there does not exist any test input that will exercise an infeasible path in the corresponding program
* Path vector
  + List of n 0’s and 1’s where each entry on the list corresponds to an edge in the flow graph
    - 0’s indicate that the corresponding edge does not lie along that path
    - 1’s indicate it does





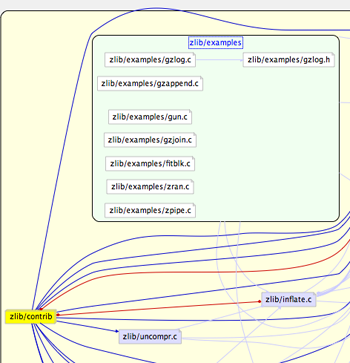


Figure -Dependency graphs illustrate the hierarchy of an architecture

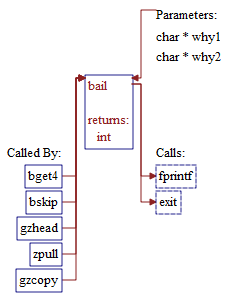


Figure -Declaration graph for a c++ method

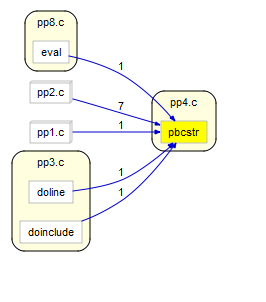


Figure -Cluster call graphs show the function call graph, organized by file

Metrics

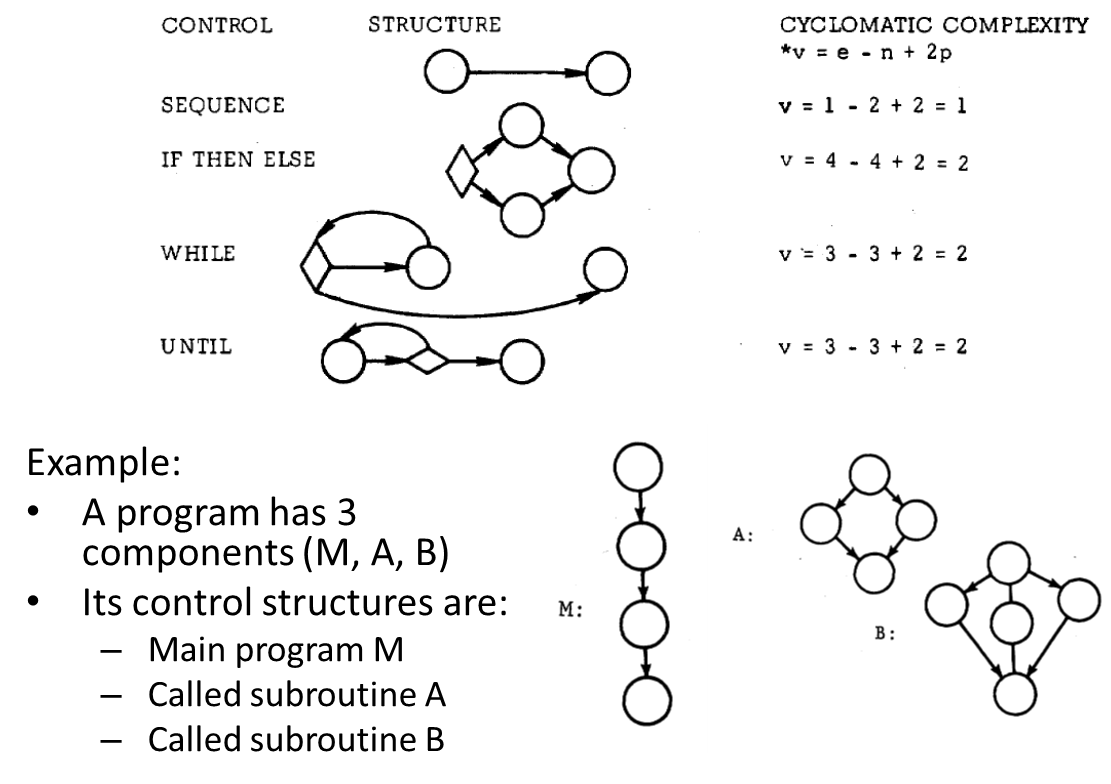
* Size
* Cyclomatic complexity
* Coupling / cohesion

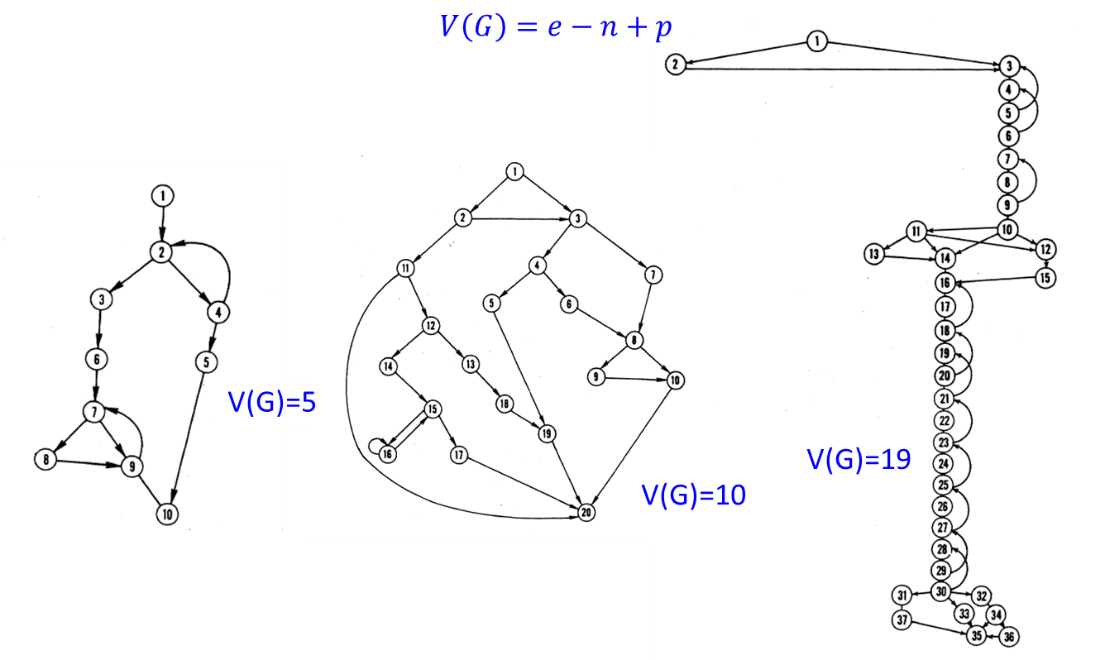
Size

* Lines of code
* Halstead metrics
  + Vocabulary
    - -number of distinct operators in a program
    - -number of distinct operands in a program
  + Length
* Volume

Cyclomatic complexity

* Graph-theoretical measure for the structural complexity of a function
* Assigns a complexity scoring mechanism the set of basis paths through a sequence of code
  + backwards branches are excluded
* Cyclomatic number,
  + -number of edges
  + -number of nodes
  + -number of components
    - i.e., separate functions that are inter-connected





### Part II

Prepare the Ubuntu VM for this class exercise by installing git, libncurses5-dev, and updating $PATH

* $ sudo apt-get install build-essential git libncurses5-dev
* Add the following line to your ~/.bashrc

|  |
| --- |
| export PATH=$PATH:/opt/scitools/bin/linux64:/opt/scitools/bin/linux64/buildspy |

* $ source ~/.bashrc
* $ mkdir ~/sandbox && cd ~/sandbox

Obtain the sample project

* $ git clone <https://github.com/devshane/zork.git> && cd zork

Edit Makefile and add the following line

|  |
| --- |
| CC = /opt/scitools/bin/linux64/buildspy/gccwrapper |

Construct the new Understand database

* $ buildspy -db zork.udb -cmd make

Start Understand and open the zork.udb project in it

Analyze all the files Project->Analyze all files

View entities (functions)

Browse the metrics for function

Add annotation

Follow call flow

Dependency graph

Cluster graph

Control flow

### Definitions

* Slicing – Analyzing a program by slices, which are sets of statements related by their flow of data; statements in a slice are not necessarily textually contiguous and may be scattered through a program

## References

* <http://www.methodsandtools.com/archive/archive.php?id=66>
* <https://cisofy.com/solutions/#lynis>
* Halstead, 1977
* Weiser, 1982
* McCabe, 1976
* SciTools Understand