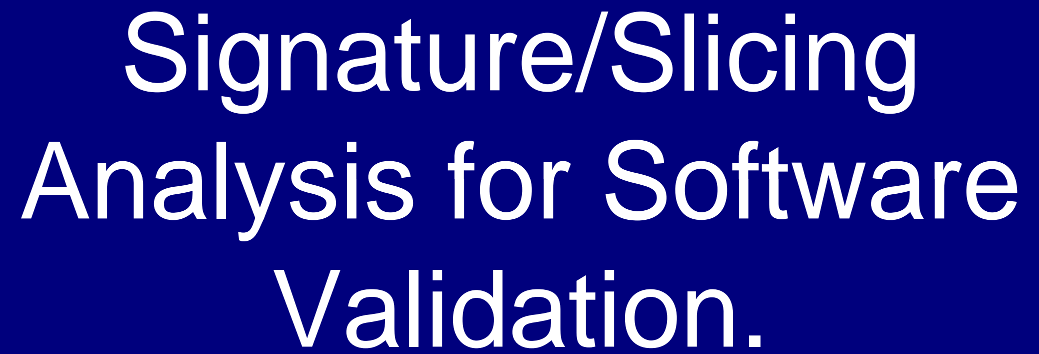
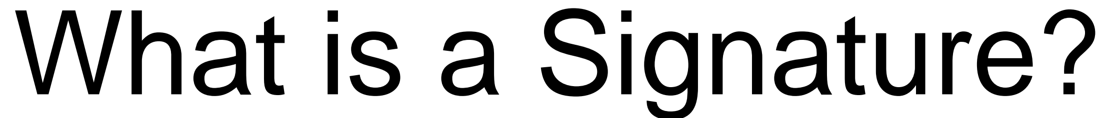

Signature/Slicing
Analysis for Software
Validation.

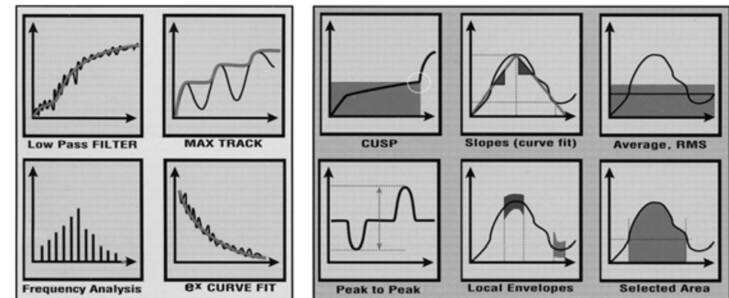


What is a Signature?

- “A distinctive mark, **characteristic**, or sound indicating identity”
- The previous definition leads to many options for Signature Analysis even limiting the topics to computer systems.

What is a Signature?

- Typically Signatures related to any electronic systems are referring to signals.
- However, in order to analyze security flaws in binaries we need to make a very specific definition related to control flow of machine code.



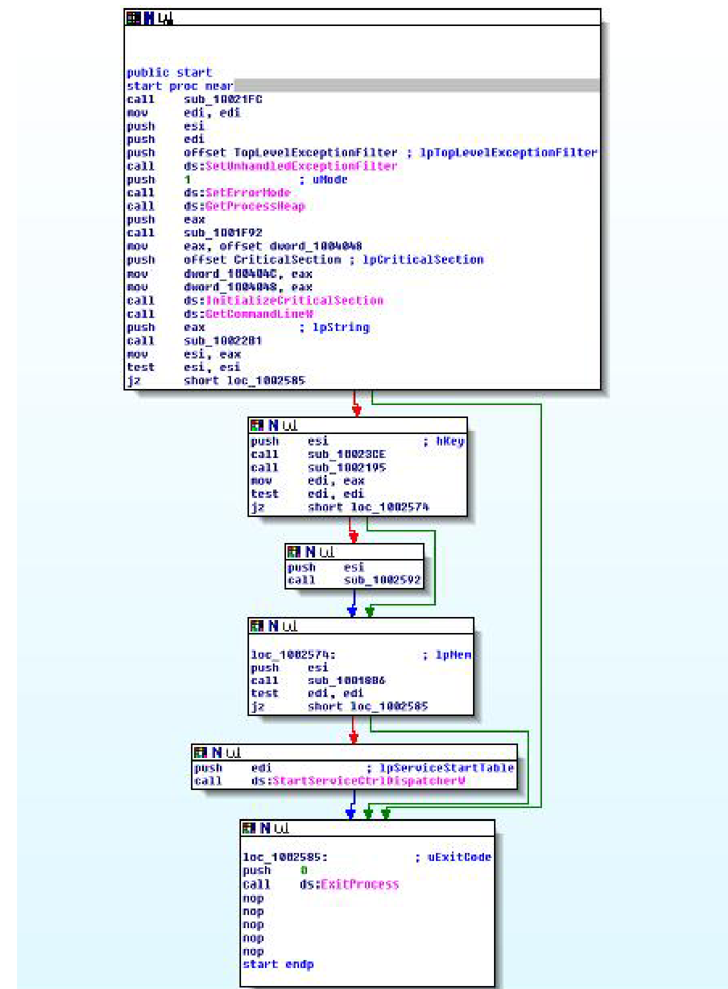
What is a Signature?

- A signature for the purposes of code detection (malicious or not) then needs to be something representing a desired behavior through static analysis.
- Therefore we can define our signature to be a unique string of machine code opcodes.

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
sub     eax, 1
mov     [ebp+arg_0], eax
mov     ecx, [ebp+arg_0]
push    ecx
push    offset aD_0     ; "%d\n"
call    printf
```

Importance of Control Flow Analysis

- Because of the nature of machine code, just searching for the string of instructions is insufficient.
- Therefore comprehension of control flow is necessary for analysis to take place.



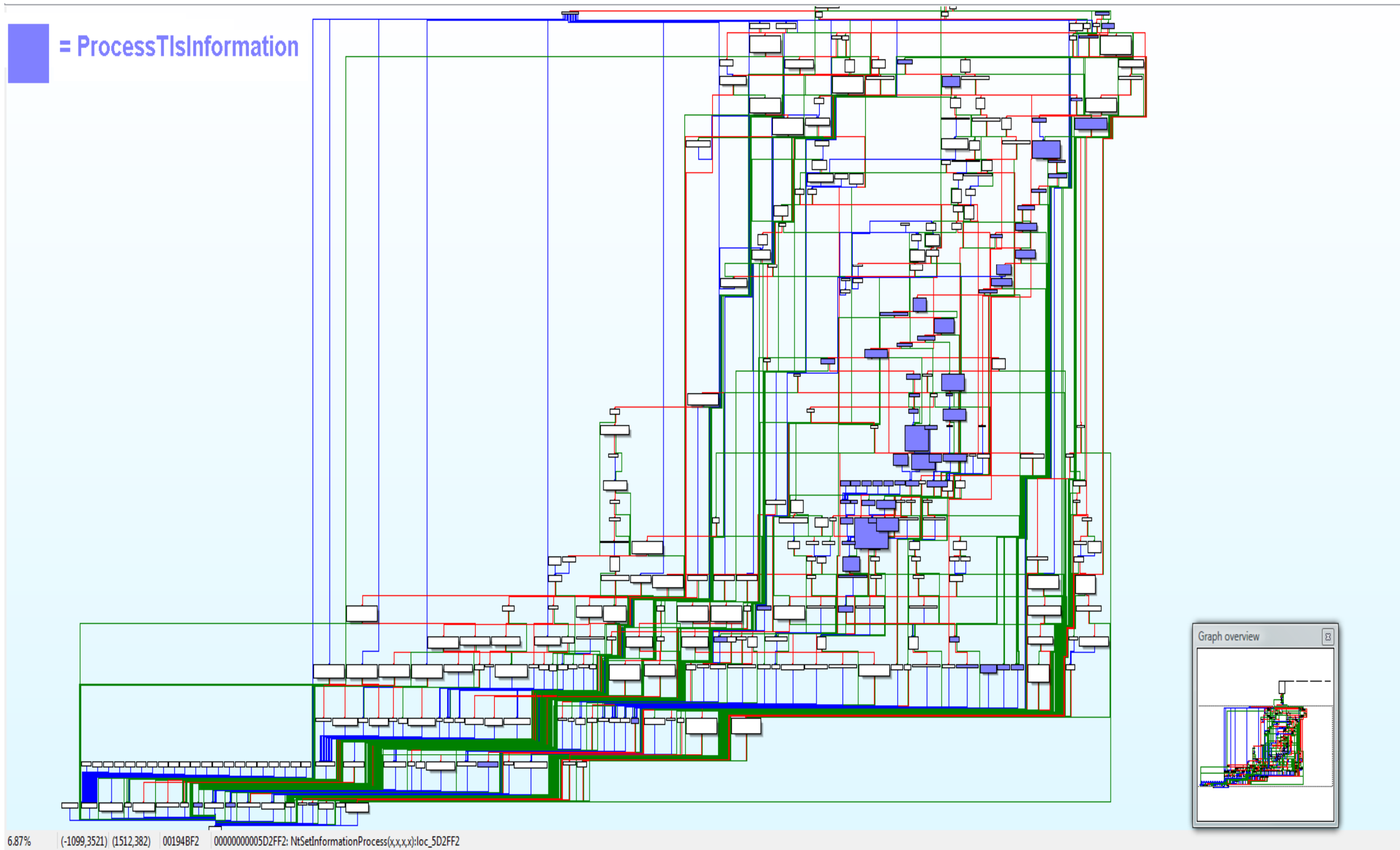
Importance of Control Flow Analysis

- Machine code can be sectioned into "Basic Blocks".
- Basic Blocks have one entry point, one exit point, and no jumps except for the last instruction.

Importance of Control Flow Analysis

- Basic Blocks directly before Basic Block ‘x’ in execution are known as “Predecessors” of ‘x’.
- Conversely, Blocks directly after ‘x’ in execution are known as “Successors” of ‘x’.
- One can now produce a Control Flow Graph by:
  - Treating Basic Blocks as nodes in a directed graph.
  - Generating directed edges as dictated by the predecessors and successors for every basic block
  - Eliminate any duplicate edges.

Importance of Control Flow Analysis



Importance of Control Flow Analysis

- A Control Flow Graph gives us the outline of every possible flow of execution.
- If Signatures Analysis is applied by checking the Control Flow Graph of a binary for a set of Signatures, we can be assured all possible execution scenarios will be analyzed.

Issues arising with Control Flow/Signature Analysis:

- NOP equivalent instructions, and groups of instructions
- Computed Jump issues with Control Flow Analysis.
  - Data Flow Analysis is needed to solve computed jumps
  - Computed jump solutions are needed for control-flow analysis.
  - Control Flow Analysis is needed for Data Flow.

Malicious code detection

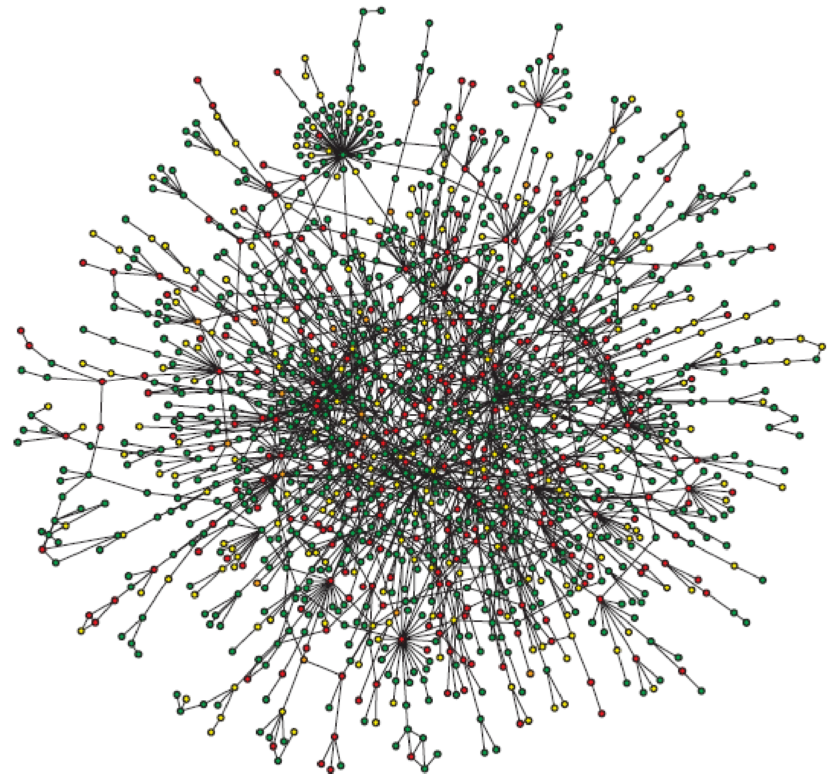
- Create a list of Signatures representing known malicious code.
- Use Signature Analysis with the malicious code Signatures to detect malicious code in an application.
- Now we know where the code is, but how can we analyze what the code might affect?

Code Slicing

- A code analysis method to determine which parts of a program may affect (backward slice) or be affected by (forward slice) variable values at some point of interest.
- When combined with Signature Analysis we can see if other parts of the code may be affected by the detected malicious code.
- http://www.cs.wisc.edu/wpis/html/

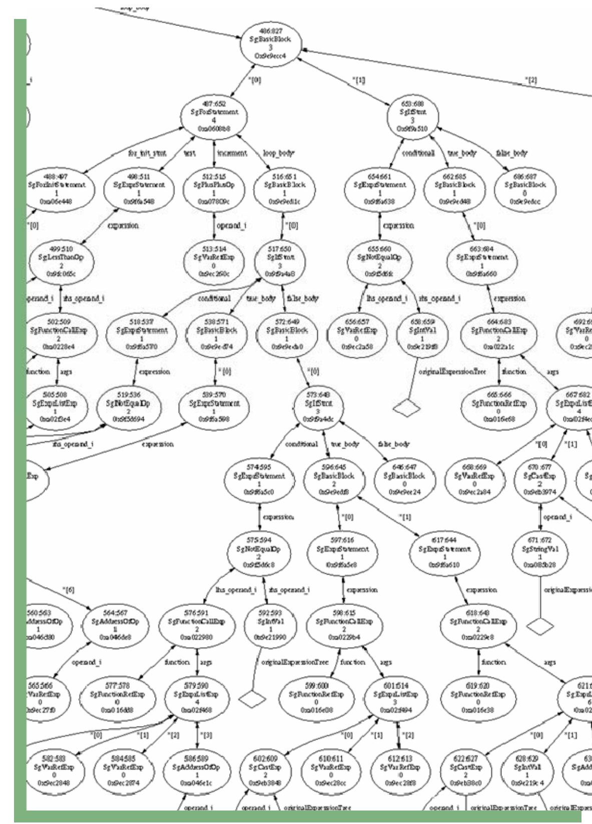
Code Slicing

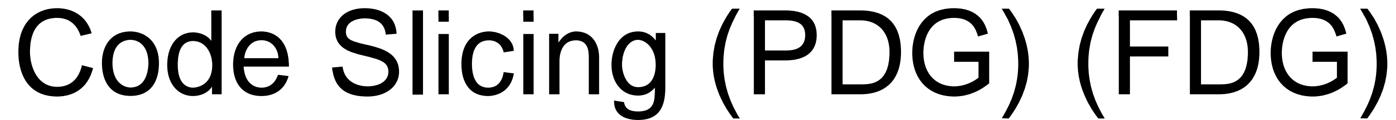
- An application of graph theory.
- Starts with a Control Flow Graph (which we have already looked at).
- Next milestone is a Program Dependence Graph. (PDG)



Code Slicing (PDG)

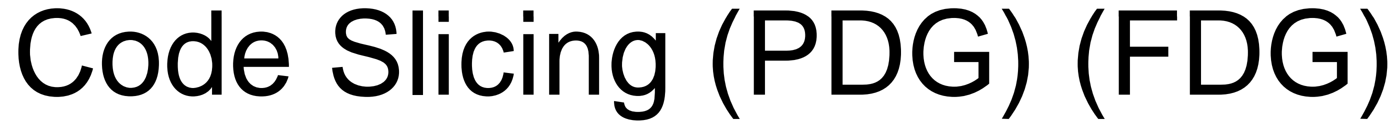
- Made up of Flow Dependence and Control Dependence Graphs.
- *Intra*procedural slicing can be accomplished with only a PDG.





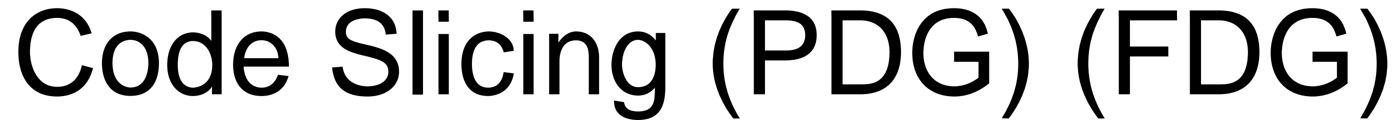
Code Slicing (PDG) (FDG)

- Flow Dependence (sometimes called Data Dependence) is sometimes referred to as the "set-use analysis" or "def-use analysis".
- For every instruction a node is created. If a variable value is set in a given node then an out edge is created to the node containing instructions in which the variable value is used.



Code Slicing (PDG) (FDG)

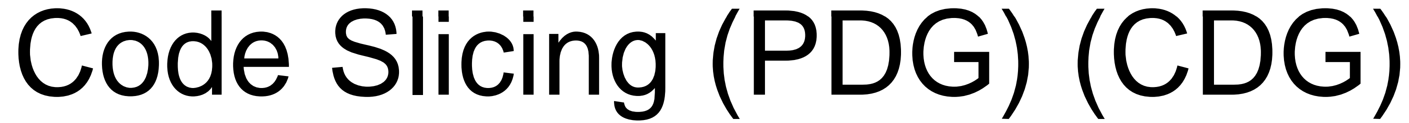
- Set – `mov 42, ax`
- Use – `add bx, ax`
- The above is also another set since the x86 add uses the 2nd register as the destination.



Code Slicing (PDG) (FDG)

```
mov bx, <address>
mov ax, [bx]
call ax
```

- These are generated by compilers for exception handlers and function pointers. Makes analysis of binaries much more difficult than analysis of source.

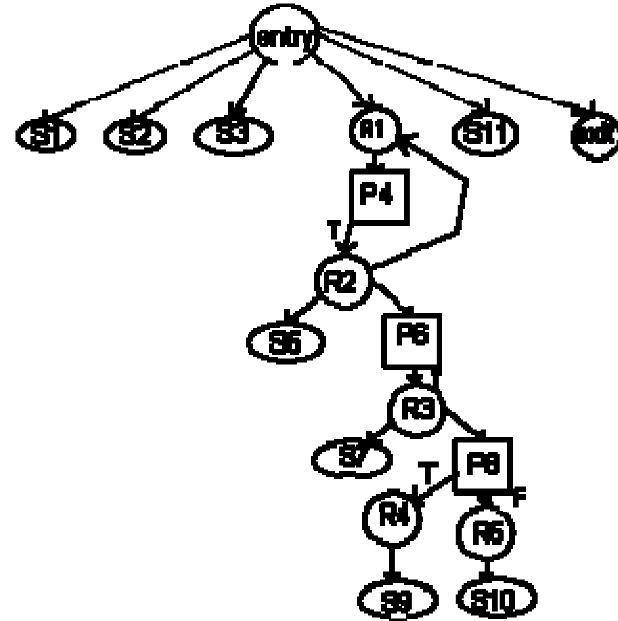


Code Slicing (PDG) (CDG)

- Control Dependence is a graph representing “reachability”.
- For every conditional statement a node is created and a “true” out edge is added to instructions reachable only if the statement is true. The same is done for false conditions.

Code Slicing (PDG) (CDG)

- Because of the lack of structure of machine code a CDG can be more difficult to generate than one from source code.
- Combining the CFG and the CDG for the entire application leads to dead-code analysis



- The combination of the FDG and the CDG is the PDG for a procedure.
- Intraprocedural Slicing is now possible.
- However, Intraprocedural Slicing is not sufficient. We need to know everything in the application that might be affected.

Code Slicing (SDG)

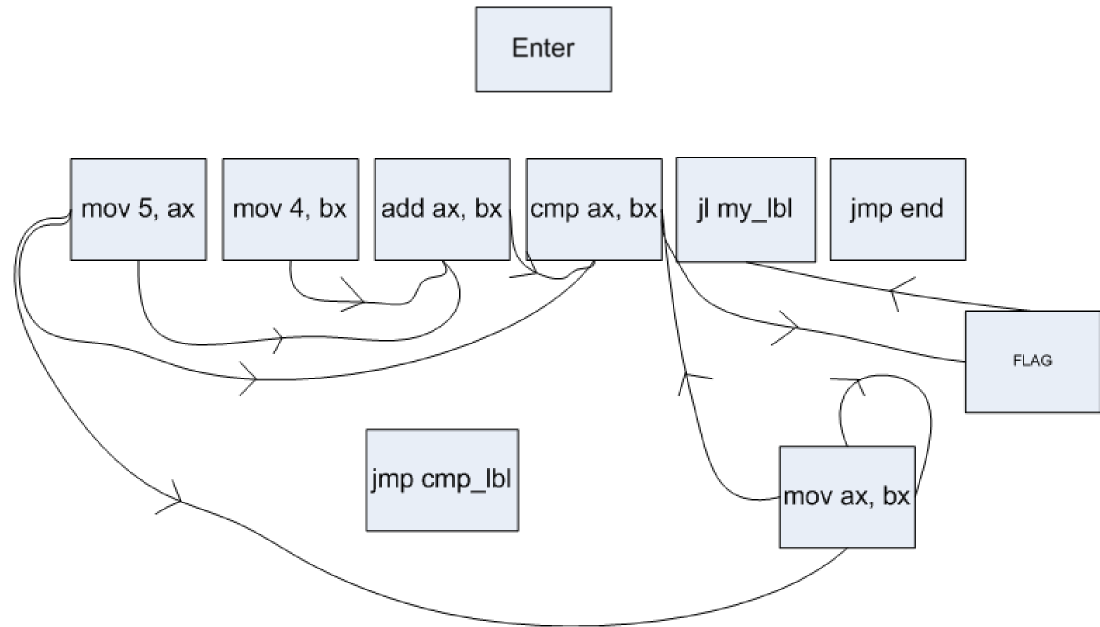
- The System Dependence Graph is a combination of all the PDGs in an application.
- Generated by
  - Creating out-edges from passed parameters to the corresponding parameters in the procedure being called.
  - Creating out-edges from the returned value of the procedure to the variable being set with the return value in the caller.

Code Slicing Example

```
          mov 5, ax
          mov 4, bx
          add ax, bx
cmp_lbl:  cmp ax, bx
          jl my_lbl
          jmp end
my_lbl:   mov ax, bx
          jmp cmp_lbl
end:
```

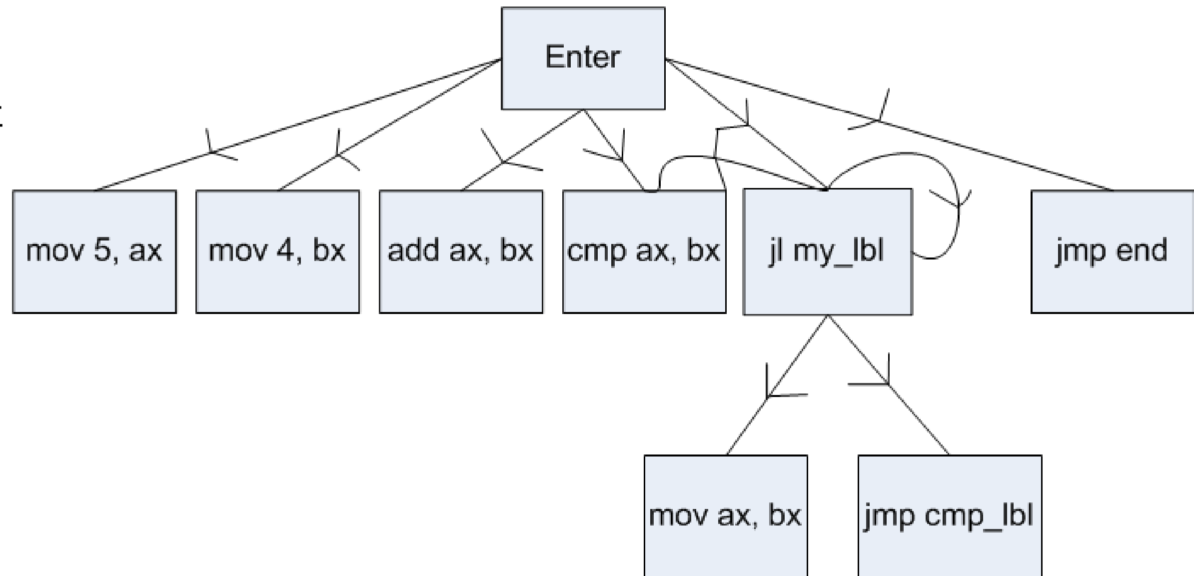
Code Slicing Example(FDG)

```
        mov 5, ax
        mov 5, bx
        add ax, bx
cmp_lbl:cmp ax, bx
        jl my_lbl
        jmp end
my_lbl:mov ax, bx
        jmp cmp_lbl
end:
```



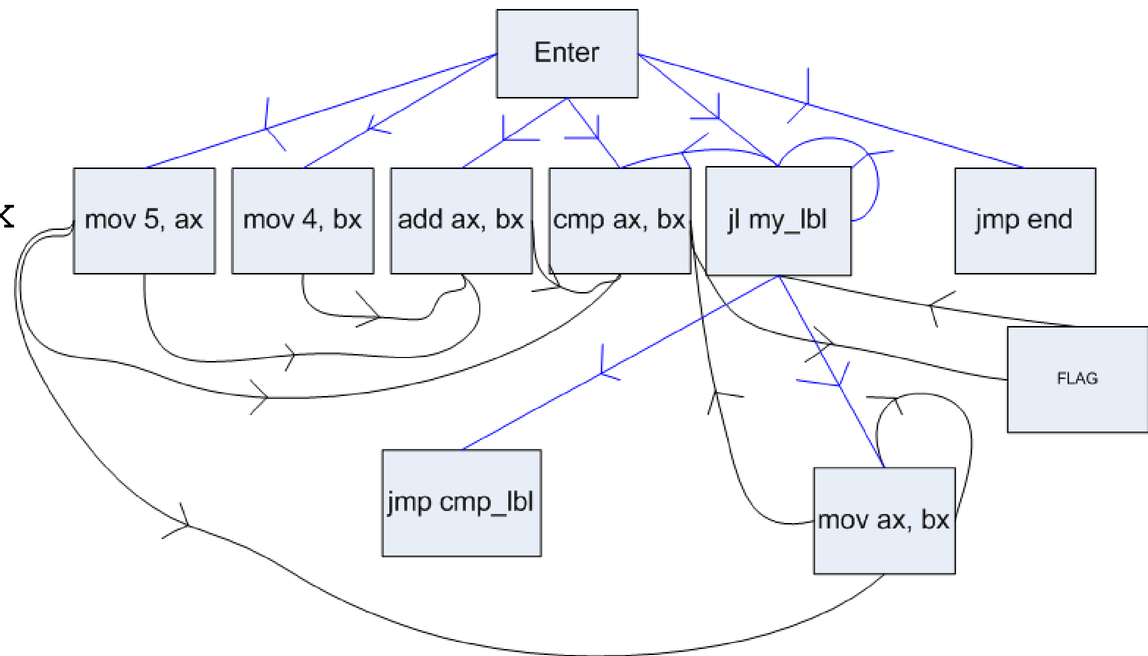
Code Slicing Example (CDG)

```
         mov 5, ax
         mov 4, bx
         add ax, bx
cmp_lbl:cmp ax, bx
         jl my_lbl
         jmp end
my_lbl:mov ax, bx
         jmp cmp_lbl
end:
```



Code Slicing Example (PDG)

```
        mov 5, ax
        mov 4, bx
        add ax, bx
cmp_lbl:cmp ax, bx
        jl my_lbl
        jmp end
my_lbl:mov ax, bx
        jmp cmp_lbl
end:
```

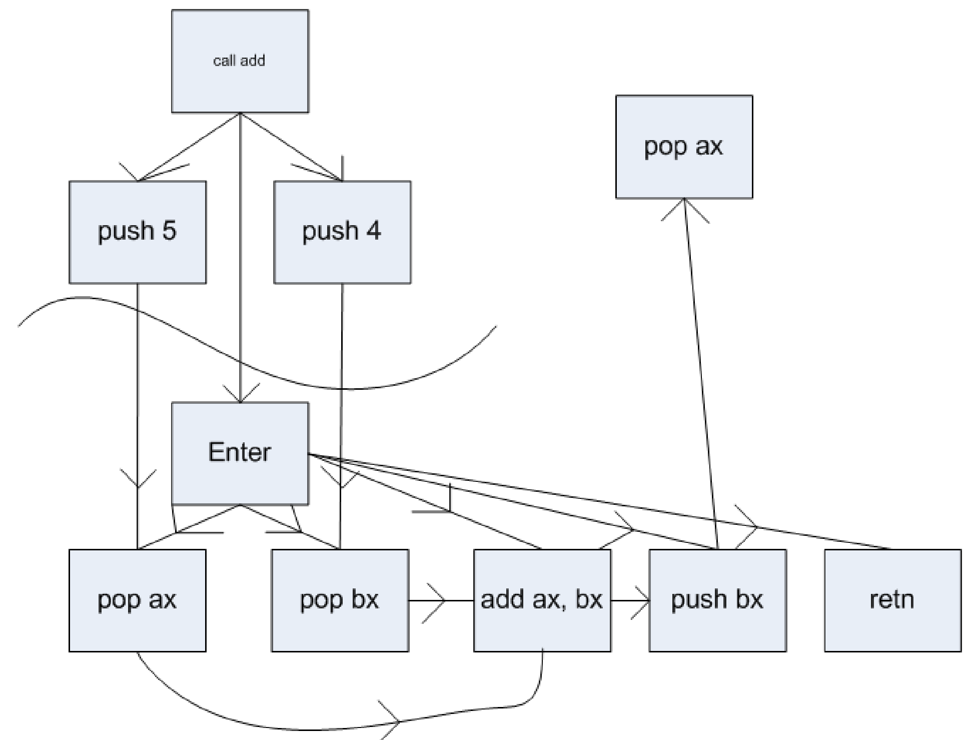


Code Slicing Example (SDG)

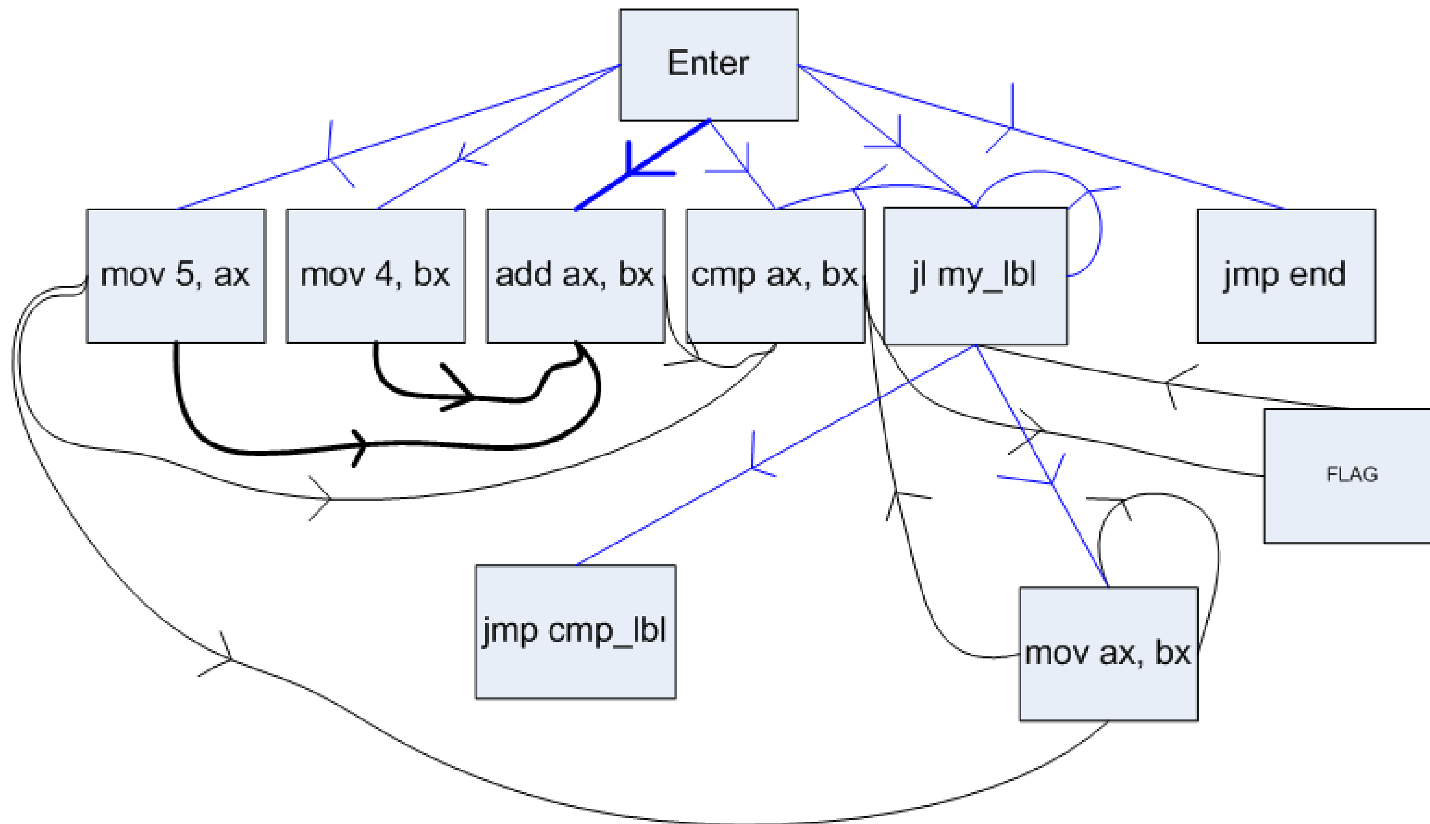
- There are no other procedures in the current example therefore the PDG shown is already a System Dependence Graph.
- But, if we were to replace the “add” instruction with an add procedure, the SDG would be created in the following way…

Code Slicing Example (SDG)

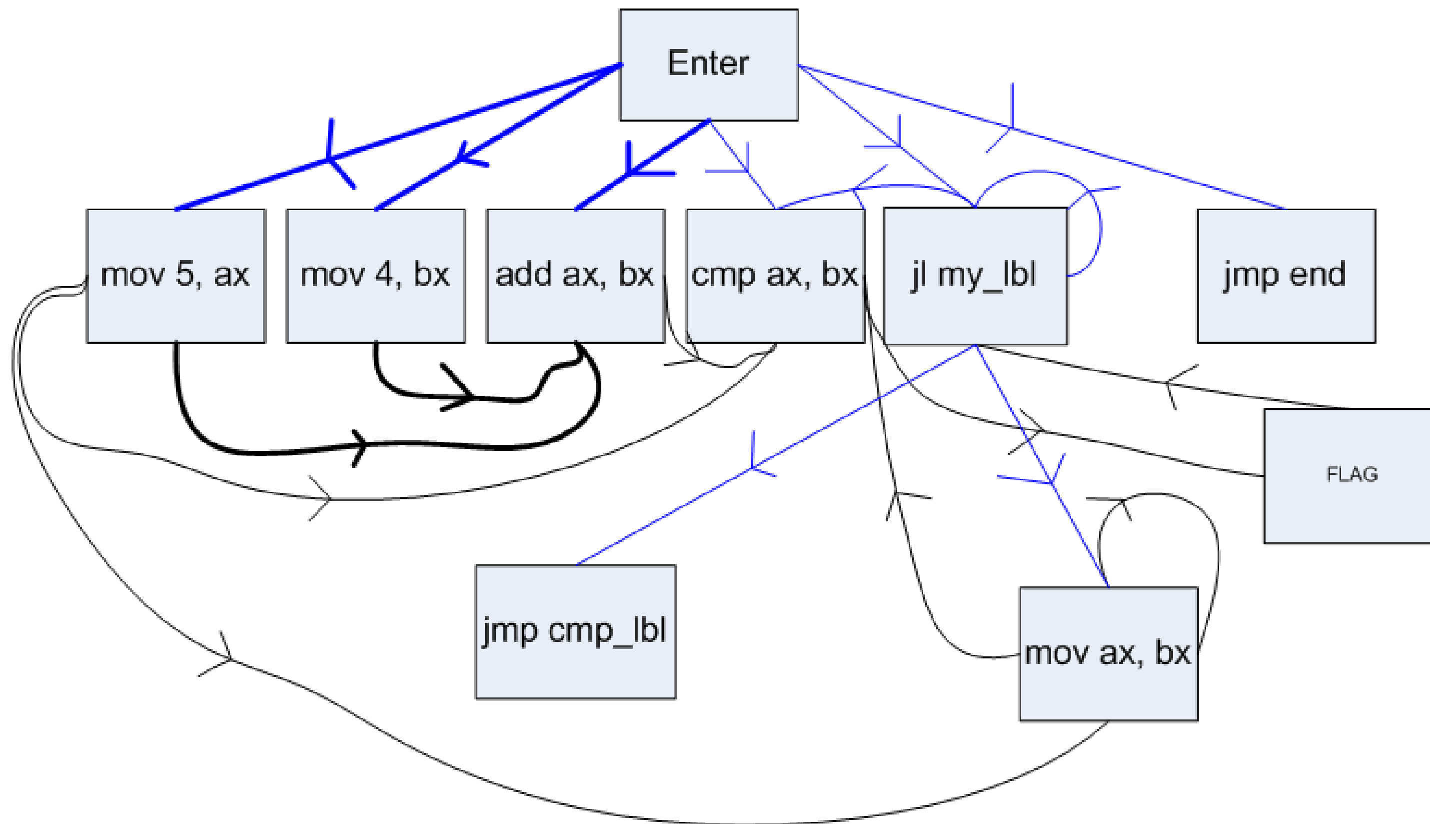
- The call has out edges to the parameters 5 and 4.
- The parameters have out edges to where they are set in the procedure (Flow Dependence)
- Also the return value has an out edge to its retrieval in the calling procedure.
- The rest is normal Flow and Control Dependence



Code Slicing Example (Backward Slice (`add ax, bx`))

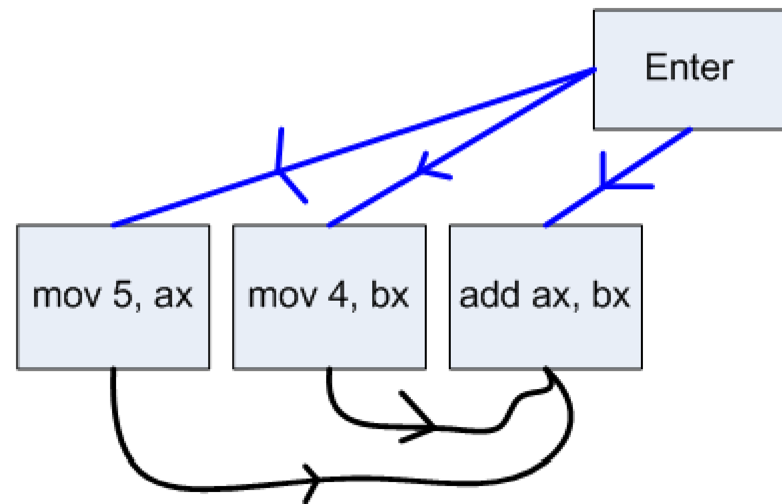


Code Slicing Example (Backward Slice (`add ax, bx`))

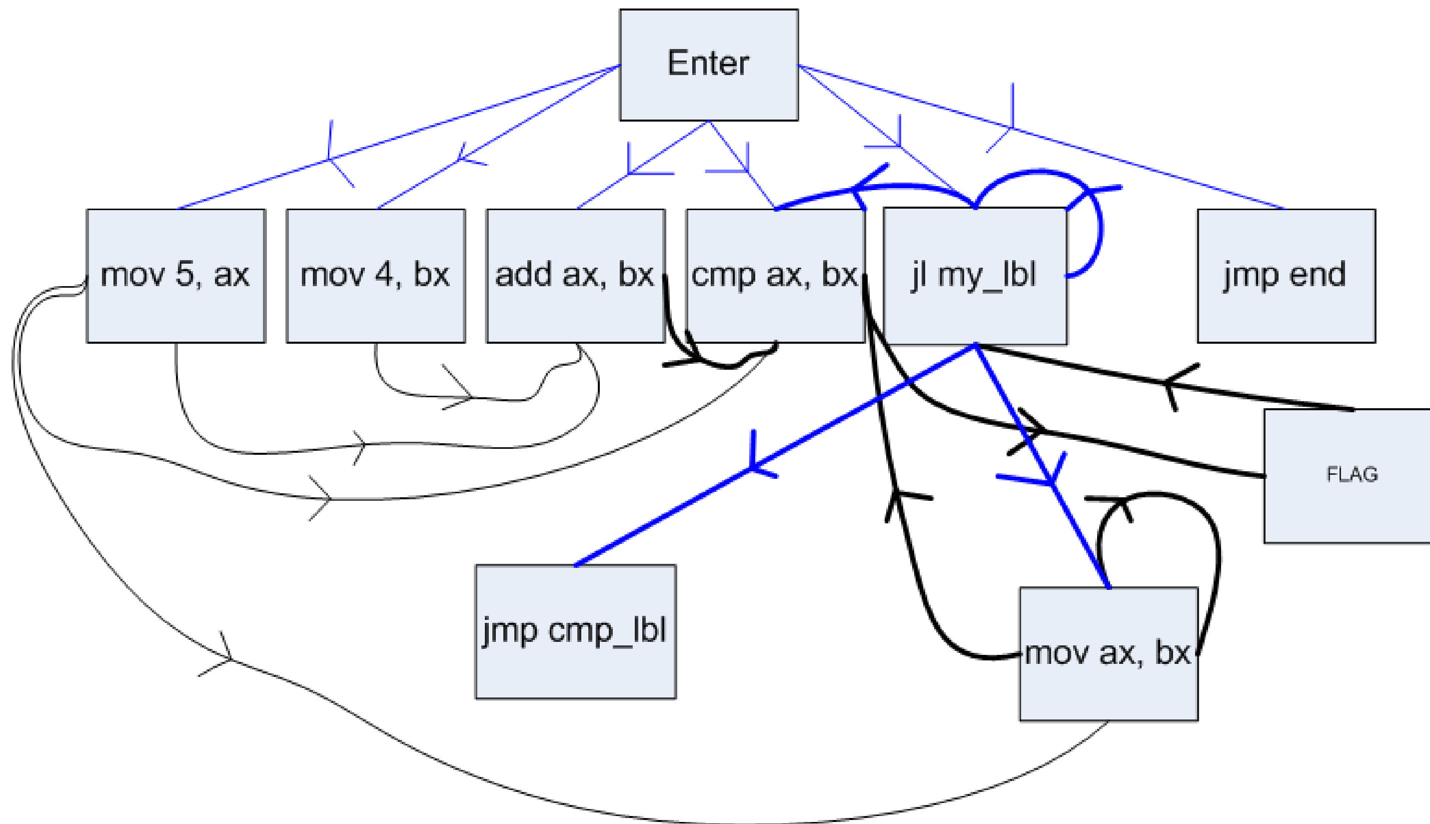


Code Slicing Example (Backward Slice (`add ax, bx`))

```
        mov 5, ax
        mov 4, bx
        add ax, bx
cmp_lbl:cmp ax, bx
        jl my_lbl
        jmp end
my_lbl:mov ax, bx
        jmp cmp_lbl
end:
```

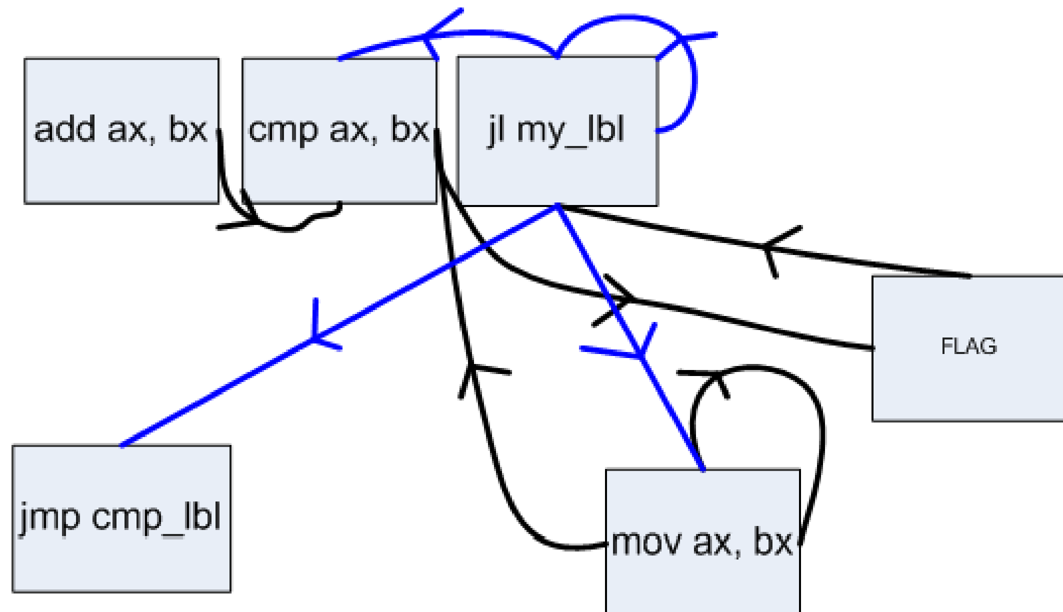


Code Slicing Example (Forward Slice (`add ax, bx`))



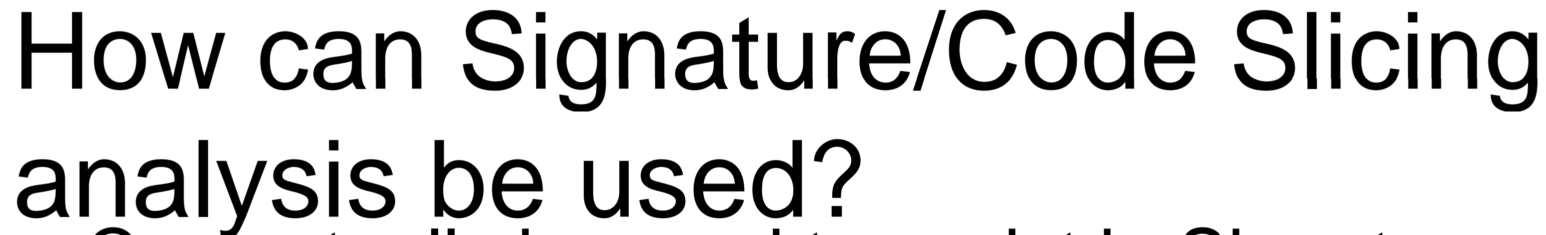
Code Slicing Example (Forward Slice (add ax, bx))

```
        mov 5, ax
        mov 4, bx
        add ax, bx
cmp_lbl:cmp ax, bx
        jl my_lbl
        jmp end
my_lbl:mov ax, bx
        jmp cmp_lbl
end:
```



Problems with Code Slicing

- Any error in analysis could quite possibly lead to a completely incorrect code slice.
- The amount of space needed to store all these graphs is very great. Following are some metrics from my colleague's Code Slicer implementation:
  - Originally a 1MB sized binary would break the 2GB limit.
  - Down to 584 megs now.
  - Used to take ~20 minutes
  - Now takes ~1 minute



How can Signature/Code Slicing analysis be used?

- Can actually be used to assist in Signature Analysis.
- The problem with NOP kinds of instructions/instruction sets can be alleviated due to proper analysis of a code slice.
- Find a possible Signature start and a possible Signature end in a binary then perform a forward and backward slice on each instruction included. If any code does not affect the rest of the Signature or binary then the code be excluded as “do nothing code.” (This would not skip important code because important code would be contained within the Signature)

How can Signature/Code Slicing analysis be used?

- The best and most desirable security is to have a resilient and hardened program from release.
- Run Code Slicing Analysis on a signature.
- Create *possible* out-edges in the appropriate places (if something is defined in the Signature, but not used etc.).

How can Signature/Code Slicing analysis be used?

- For Flow Dependence the possible out-edges would tell you which registers used in a certain way might be affected by a known security threat.
- For Control Dependence the possible out-edges would tell you where else in the application a given Signature might cause a jump to or if a section of code might be skipped entirely.

How can Signature/Code Slicing analysis be used?

- Note: Analyzing Flow Dependence and Control Dependence in the given way requires to not just know the algorithm for Code Slicing Analysis but to truly understand what a Code Slice represents.

How can Signature/Code Slicing analysis be used?

- Code Slicing Analysis gives some additional insight into the behavior of the malicious code represented by the Signature.
- The behavior can then be caught by a separate thread or process which understands what threats are possible in what locations and to alert and/or deal with the problem.

Information and Questions

- Adam Dugger
  - adugger@arxan.com
  - www.arxan.com
- Any questions?