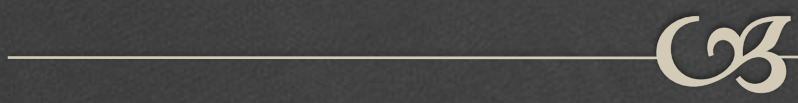
Control Flow Analysis



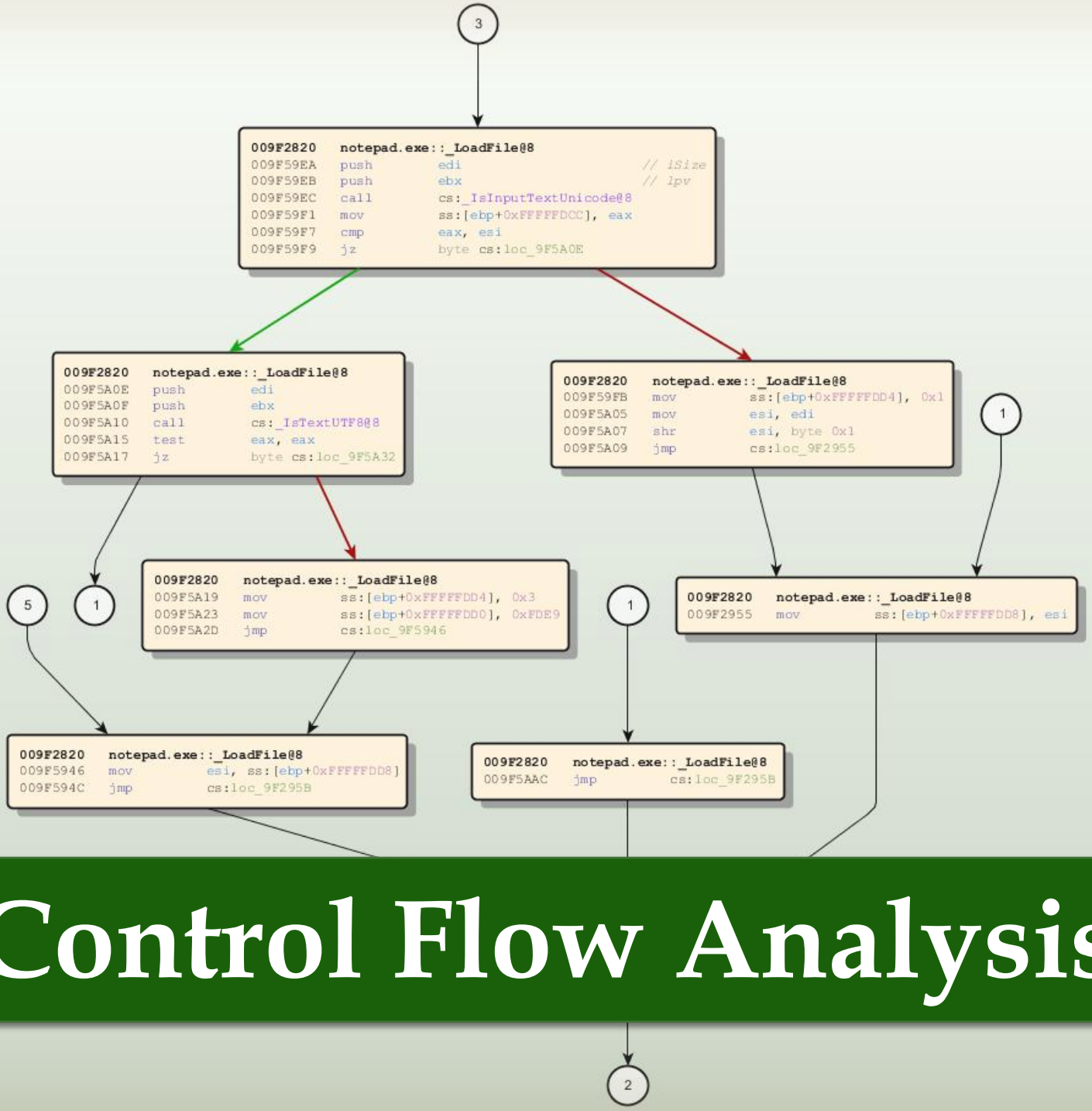
Edgar Barbosa

H2HC 2011

São Paulo - Brazil

Who am I?

- Edgar Barbosa
- Senior Security Researcher at COSEINC (Singapore)
- One of the developers of Blue Pill, a hardware-based virtualization rootkit. Also presented a way to detect this type of rootkit.
- Discovered the Windows kernel KdVersionBlock data structure used for some forensic tools.
- Focus: RCE, Windows Internals, Virtualization and Program Analysis.
- Currently working on the COSEINC SMT Project, which aims to automate the bug finding process with the help of SMT solvers. The current presentation is part of the research done for the SMT project.

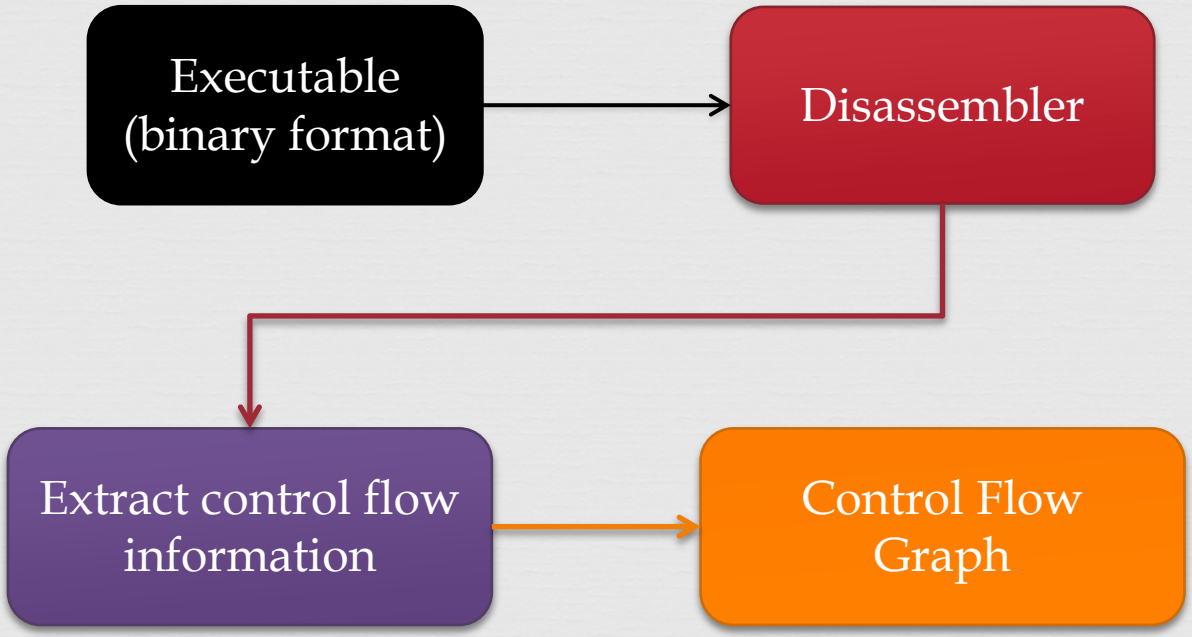


Control Flow Analysis

Control Flow Analysis

- Control Flow Analysis (CFA)
- Static analysis technique to discover the *hierarchical flow of control* within a procedure (function).
- Analysis of all possible *execution paths* inside a program or procedure.
- Represents the control structure of the procedure using ***Control Flow Graphs***.
- Compiler theory - optimization
- The focus of this presentation is to demonstrate CFA for Reverse Code Engineering, where the source code isn't available.

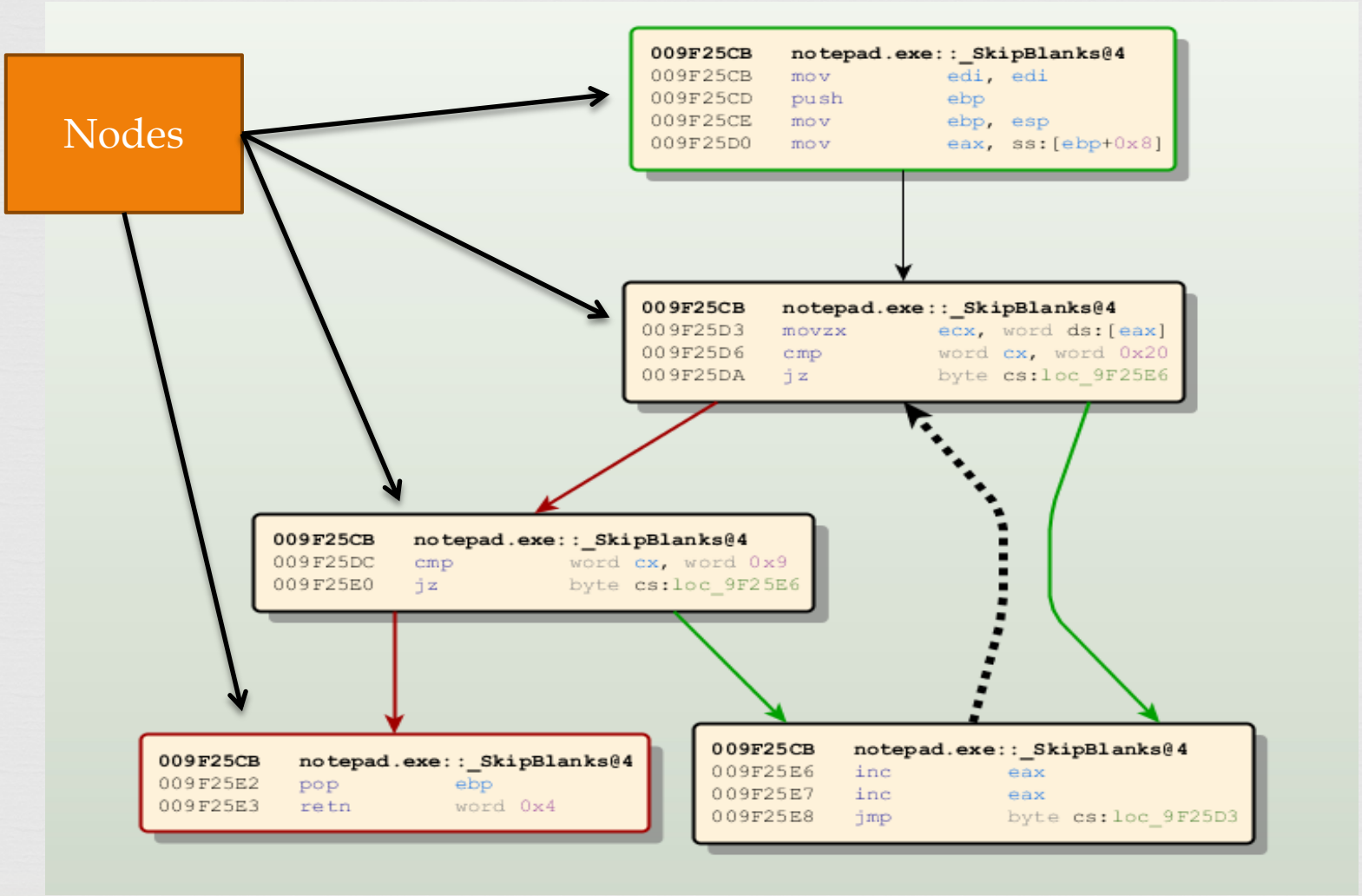
RCE and CFA


Executable
(binary format)
Disassembler
Extract control flow
information
Control Flow
Graph

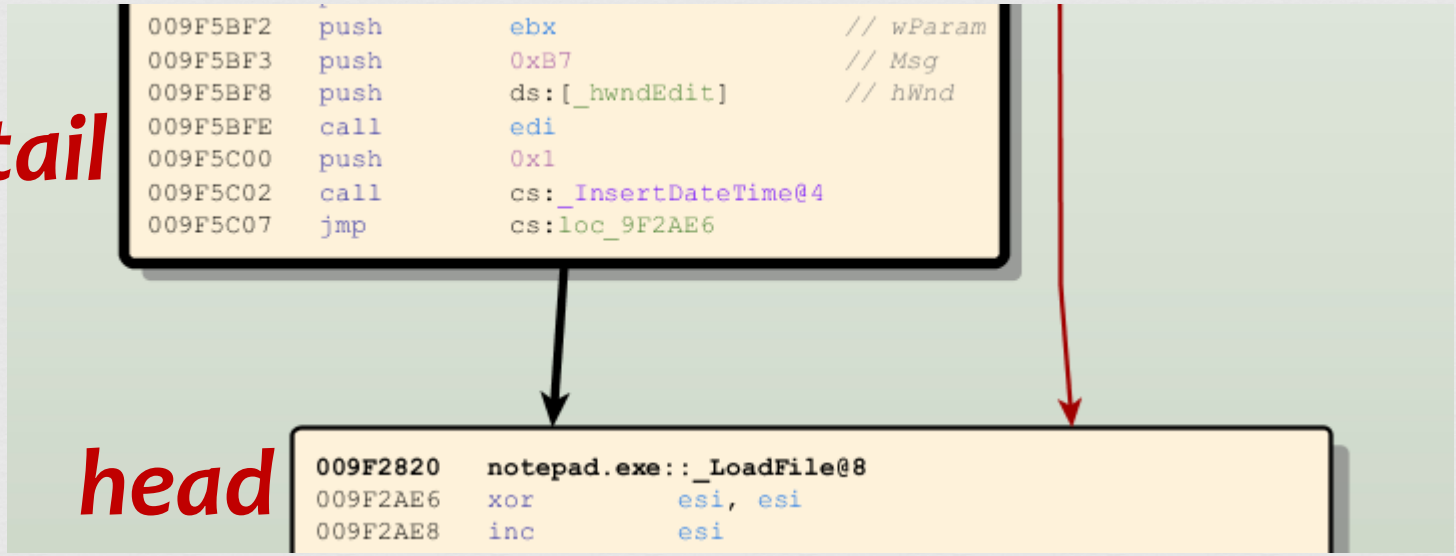
What is a CFG?

- A Control Flow Graph (CFG) is a directed graph G(V;E) which consists of a set of vertices (nodes)V, and a set of edges E, which indicate possible flow of control between nodes
- Or, is a **directed graph** that represents a superset of ***all possible execution paths*** of a procedure.
- Graph nodes represents objects called *Basic Blocks (BB)*

CFG

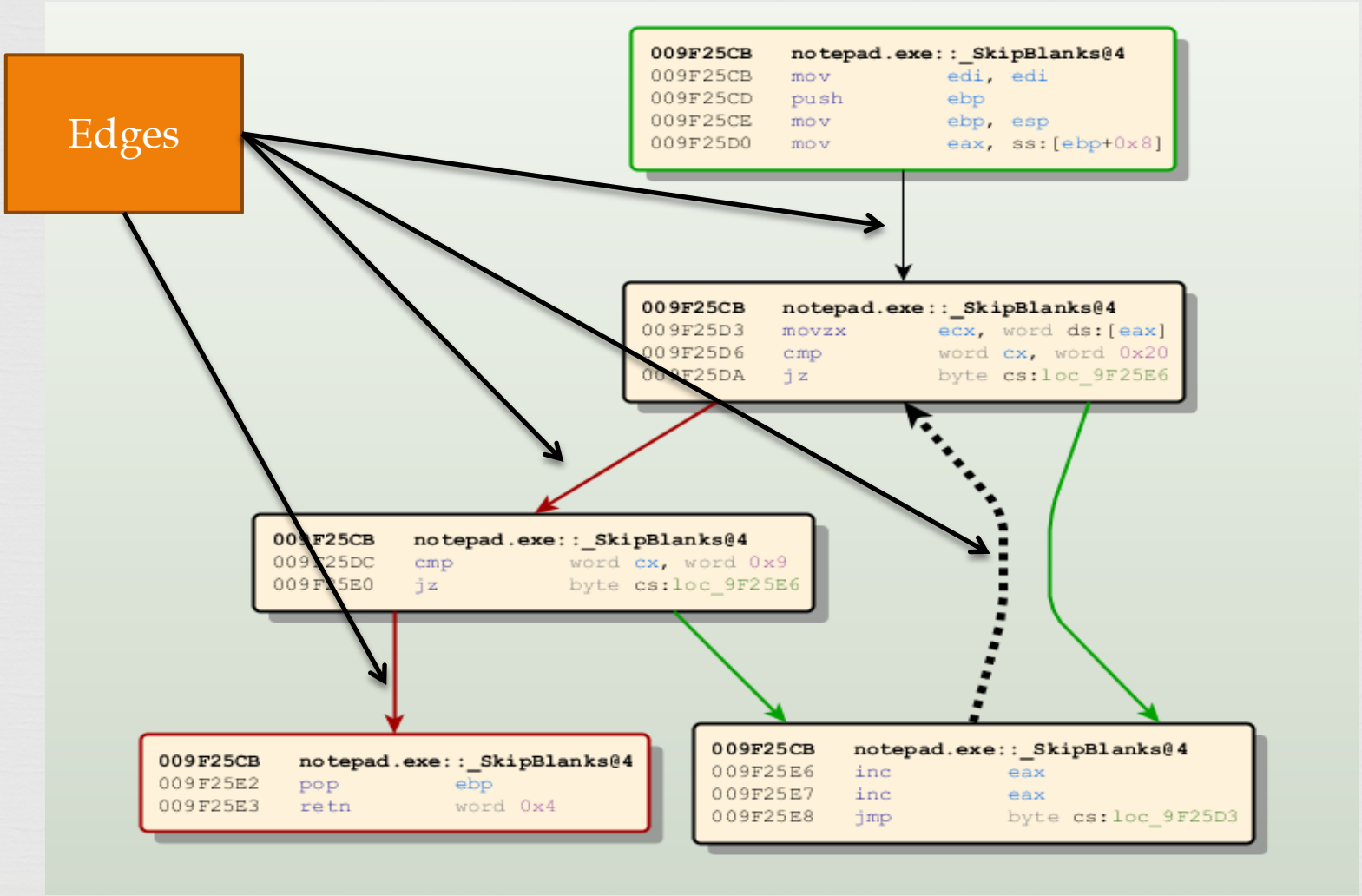

Nodes
009F25CB notepad.exe::_SkipBlanks@4
009F25CB mov edi, edi
009F25CD push ebp
009F25CE mov ebp, esp
009F25D0 mov eax, ss:[ebp+0x8]
009F25CB notepad.exe::_SkipBlanks@4
009F25D3 movzx ecx, word ds:[eax]
009F25D6 cmp word cx, word 0x20
009F25DA jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25DC cmp word cx, word 0x9
009F25E0 jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25E2 pop ebp
009F25E3 retn word 0x4
009F25CB notepad.exe::_SkipBlanks@4
009F25E6 inc eax
009F25E7 inc eax
009F25E8 jmp byte cs:loc_9F25D3

Edges



tail → head

CFG


Edges
009F25CB notepad.exe::_SkipBlanks@4
009F25CB mov edi, edi
009F25CD push ebp
009F25CE mov ebp, esp
009F25D0 mov eax, ss:[ebp+0x8]
009F25CB notepad.exe::_SkipBlanks@4
009F25D3 movzx ecx, word ds:[eax]
009F25D6 cmp word cx, word 0x20
009F25DA jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25DC cmp word cx, word 0x9
009F25E0 jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25E2 pop ebp
009F25E3 retn word 0x4
009F25CB notepad.exe::_SkipBlanks@4
009F25E6 inc eax
009F25E7 inc eax
009F25E8 jmp byte cs:loc_9F25D3

BinNavi

- Views
- Nodes
- Edges

CFG properties

- In the CFA literature the algorithms assume the following CFG properties:
  - Unique Start node (Entry node)
  - All the nodes of must be reachable from the START node.
  - Unique Exit node
- Real-world:
  - Easy to find multiple exit nodes (RETURN) on the disassembly of a function
- Create a new exit node, add it to the graph and modify the return instructions to jump to the new node.

BB identification

- In general, the problem of discovering all the possible execution paths of a code is *undecidable*. (cf. *Halting problem)*.
- First step for CFG reconstruction is to identifiy all the basic blocks.
- A basic block is a *maximal sequence* of instructions that can be entered only at the first of them and exited only from the last of them

Basic Block (BB)

Basic Blocks

- First instruction of a BB (the *leader* instruction):
  1. The entry point of the routine
  2. The target of a branch instruction
  3. The instruction immediately following a branch
- Although CALL is a branch instruction, the target function is assumed to *always* return and therefore it is allowed in the middle of a BB.
- To build the BB's we need to identify all the *leader* instructions. This requires the disassembly of the instructions.
- Two disassembly algorithms

1. Linear Sweep

- A linear sweep algorithm starts with the first byte in the code section and proceeds by decoding each byte until an illegal instruction is encountered [a]

```
>> 8B FF 55 8B EC 8B 45 08

8B FF       mov  edi, edi
55          push ebp
8B EC       mov  ebp, esp
8B 45 08    mov  eax, [ebp+8]
```

2. Recursive Traversal

- Linear sweep algorithm doesn't take into account the control flow behaviour of some instructions.

```
>> EB 01 FF 8B 45 FC


   EB 01    jmp short 0x401020
   FF       ???   ;invalid
```

- Recursive traversal disassemblers interpret branch instructions in the program to translate only those bytes which can actually be *reached* by control flow. [b]

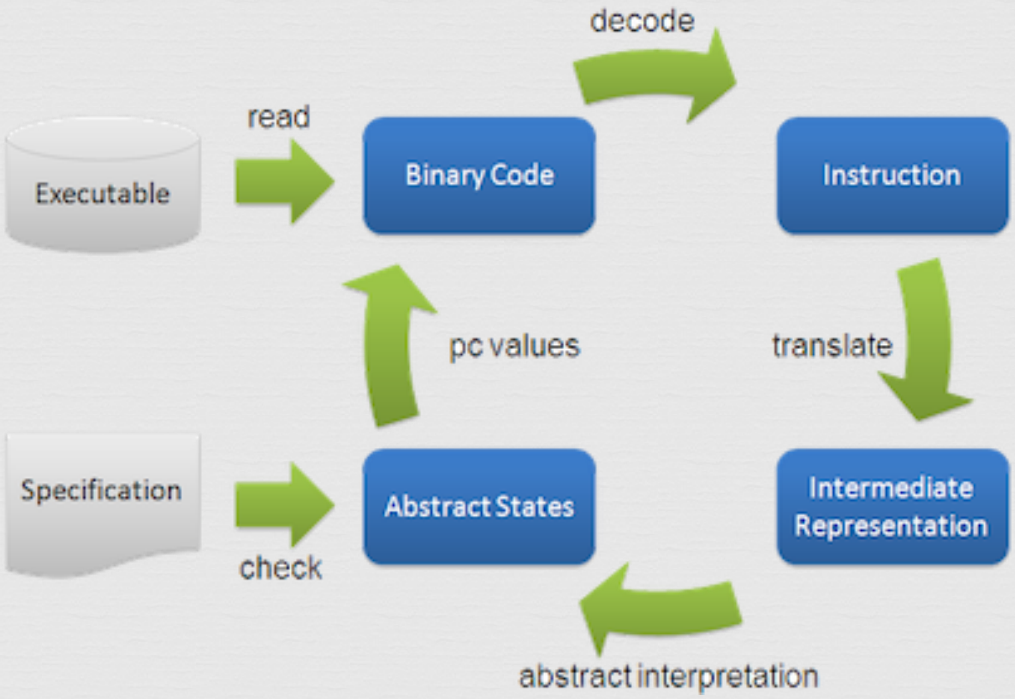
2. Recursive Traversal


```
EB 01     jmp short 0x401020
FF        ???   (UNREACHABLE)
8B 45 FC mov eax, dword ptr ss:[ebp-4]
```

State-of-art CFG reconstruction

- Once identified the basic blocks, the CFG construction is done after the addition of the edges.
- CFG construction is especially difficult when the code includes *indirect calls*. (`call dword ptr[eax]`)
- State-of-art CFG construction available is the open-source Jakstab tool (Java Toolkit for Static Analysis of Binaries) from Johannes Kinder.
- Provides better results than IDAPro.

Jakstab[d]


decode
read
Executable
Binary Code
Instruction
pc values
translate
Specification
Abstract States
Intermediate Representation
check
abstract interpretation

Self-modifying code

Control Flow Analysis

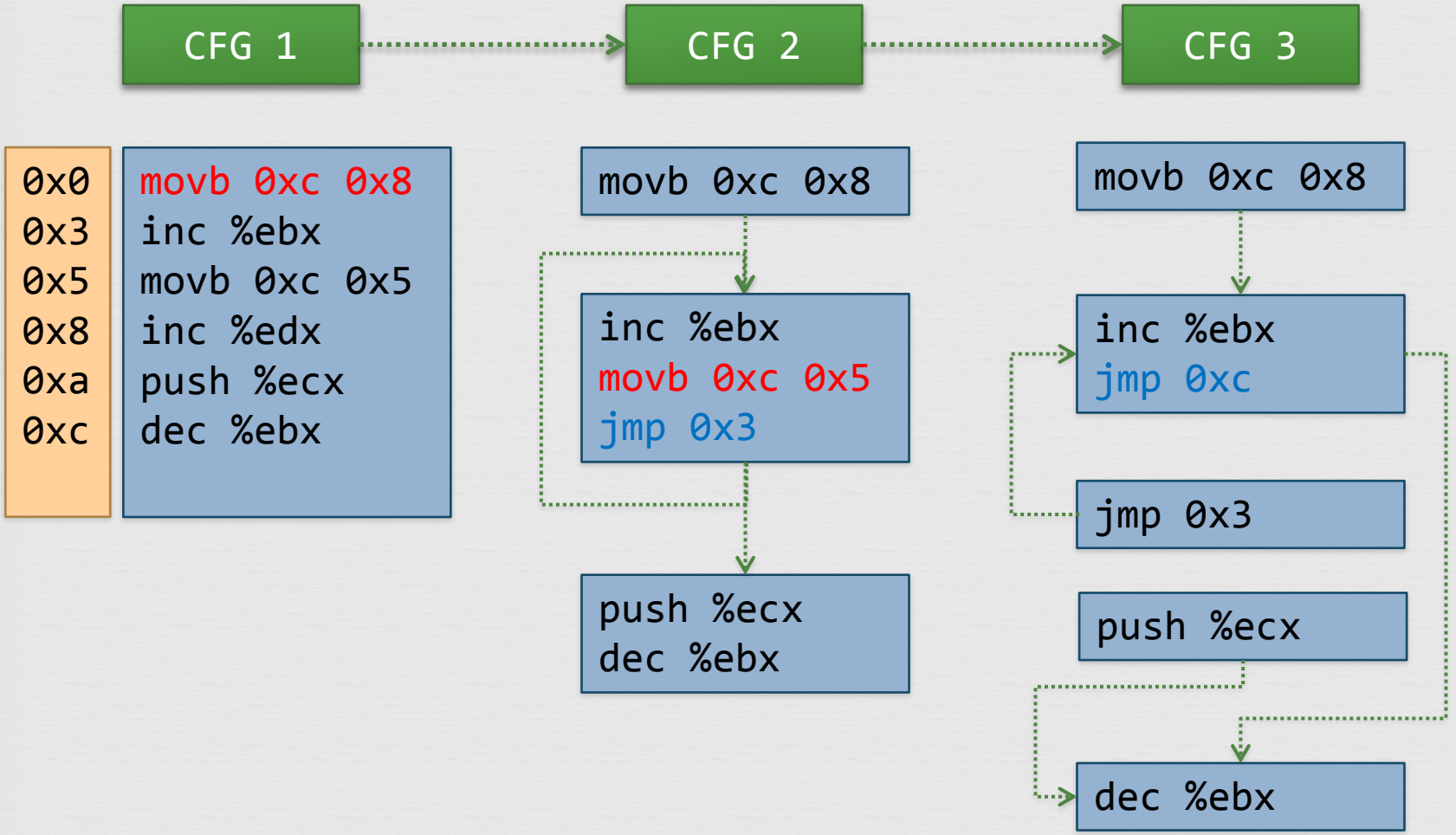
Self-modifying code

- Consider the following example[c] (*not real x86 opcodes*)

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01
0x5	movb 0xc 0x5	c6 0c 05
0x8	inc %edx	40 03
0xa	push %ecx	ff 02
0xc	dec %ebx	48 01

- A linear sweep or recursive traversal algorithm execution on the above code would result in a single Basic Block (single entry/single exit/no branches)

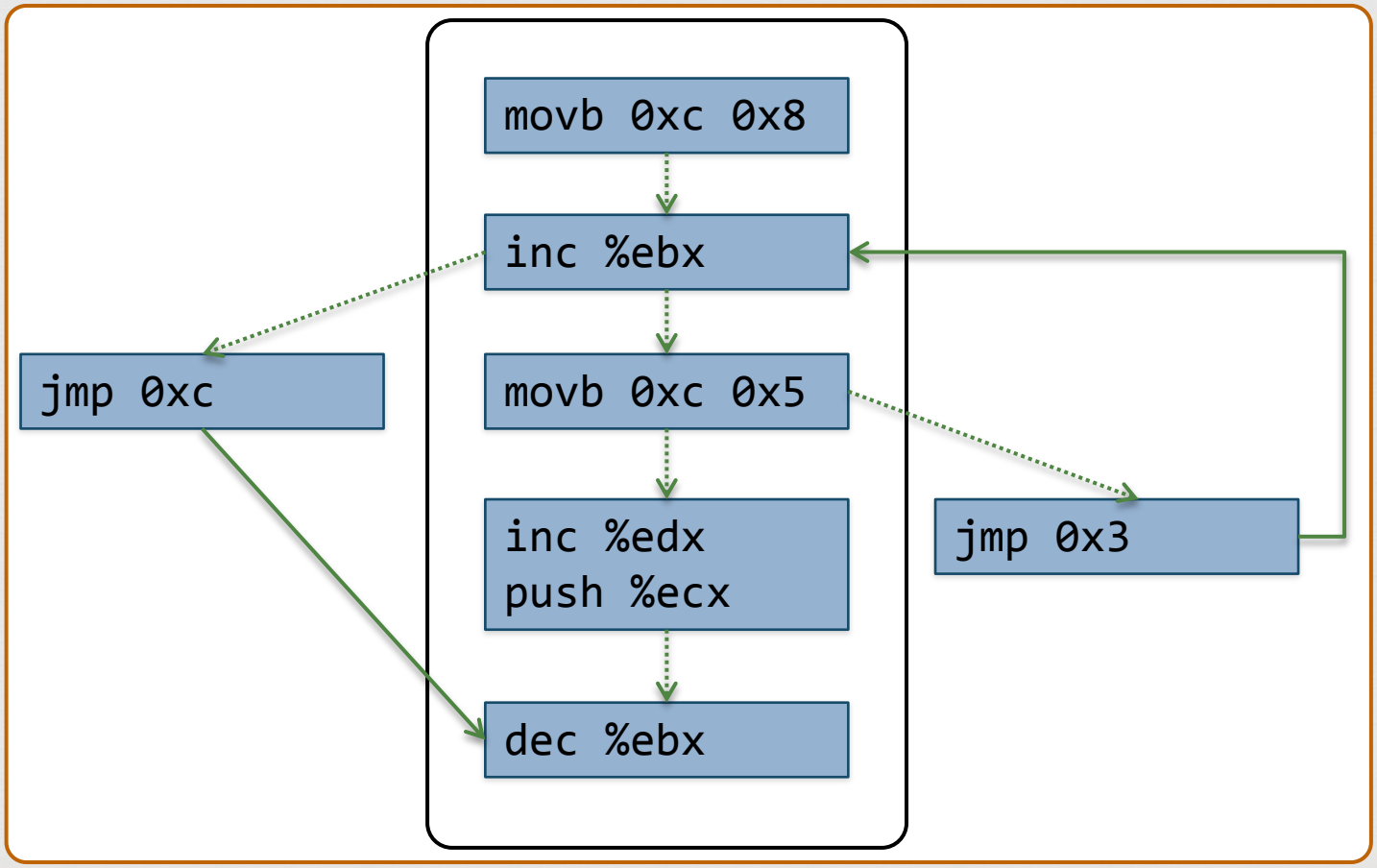
SMC

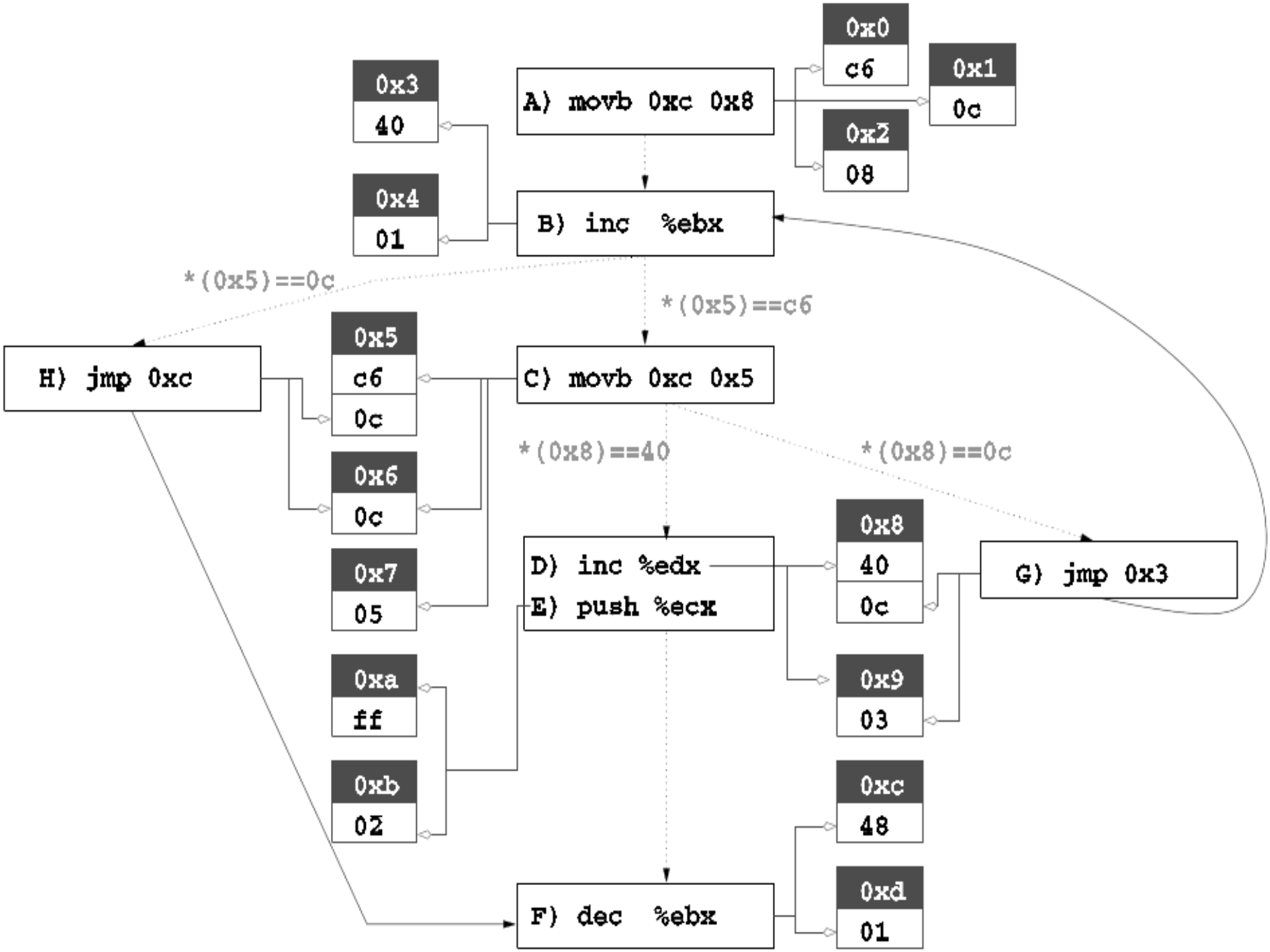

CFG 1
CFG 2
CFG 3
0x0
0x3
0x5
0x8
0xa
0xc
movb 0xc 0x8
inc %ebx
movb 0xc 0x5
inc %edx
push %ecx
dec %ebx
movb 0xc 0x8
inc %ebx
movb 0xc 0x5
jmp 0x3
push %ecx
dec %ebx
movb 0xc 0x8
inc %ebx
jmp 0xc
jmp 0x3
push %ecx
dec %ebx

SE-CFG

- State-Enhanced Control Flow Graph (SE-CFG)
- CFG augmented with extensions to support SMC.
- Allows the use of control flow analysis algorithms for SMC.
- “A Model for Self-Modifying Code”
- Codebyte extensions – Codebyte conditional edges
- Implemented in a link-time binary rewriter: Diablo.
- It can be downloaded from
  - http://www.elis.ugent.be/diablo

SMC - CFG


movb 0xc 0x8
inc %ebx
jmp 0xc
movb 0xc 0x5
inc %edx
push %ecx
jmp 0x3
dec %ebx


0x0
c6
0x1
0c
0x2
08
0x3
40
0x4
01
A) movb 0xc 0x8
B) inc %ebx
*(0x5)==0c
*(0x5)==c6
H) jmp 0xc
0x5
c6
0c
C) movb 0xc 0x5
0x6
0c
0x7
05
*(0x8)==40
*(0x8)==0c
D) inc %edx
E) push %ecx
0x8
40
0c
G) jmp 0x3
0x9
03
0xa
ff
0xb
02
0xc
48
0xd
01
F) dec %ebx

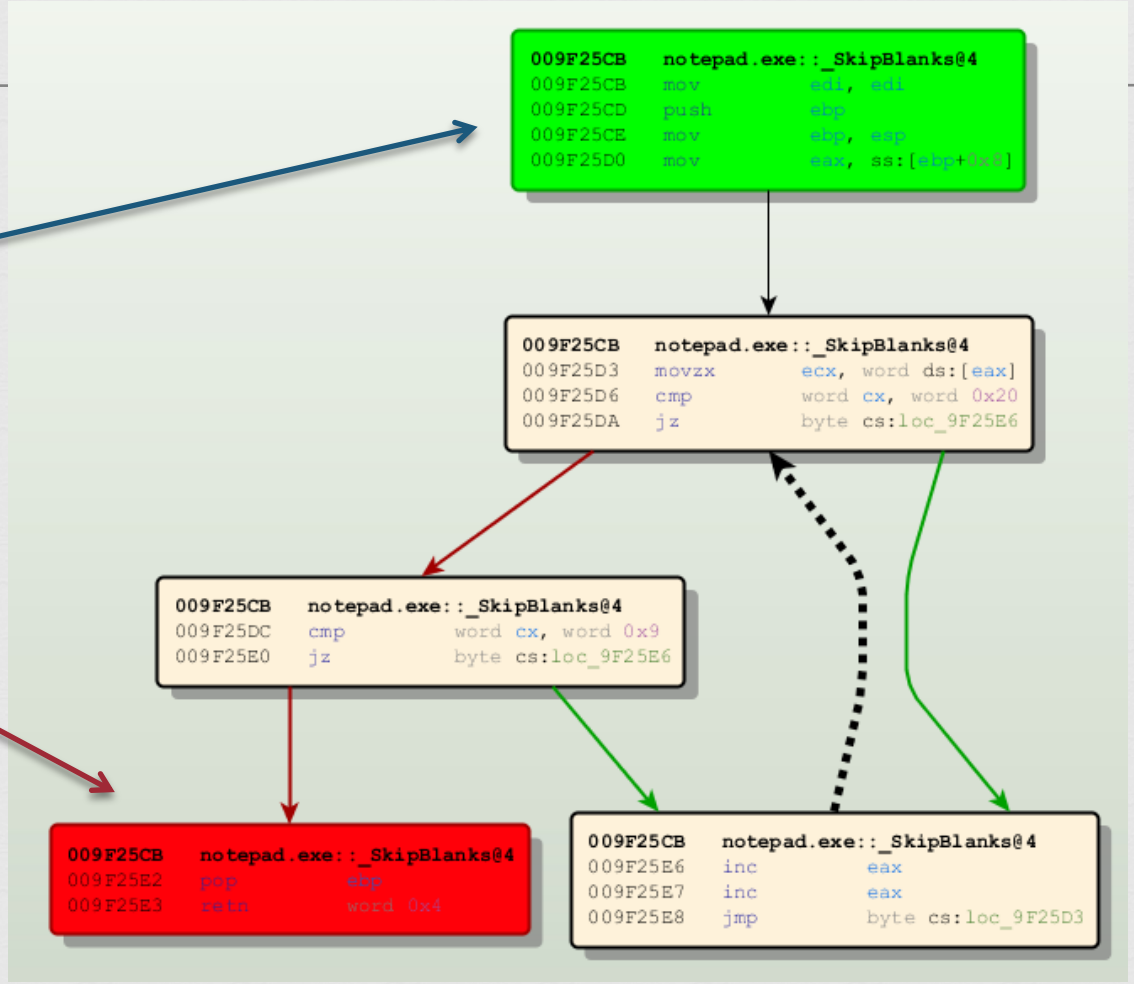
Dominators

Control Flow Analysis

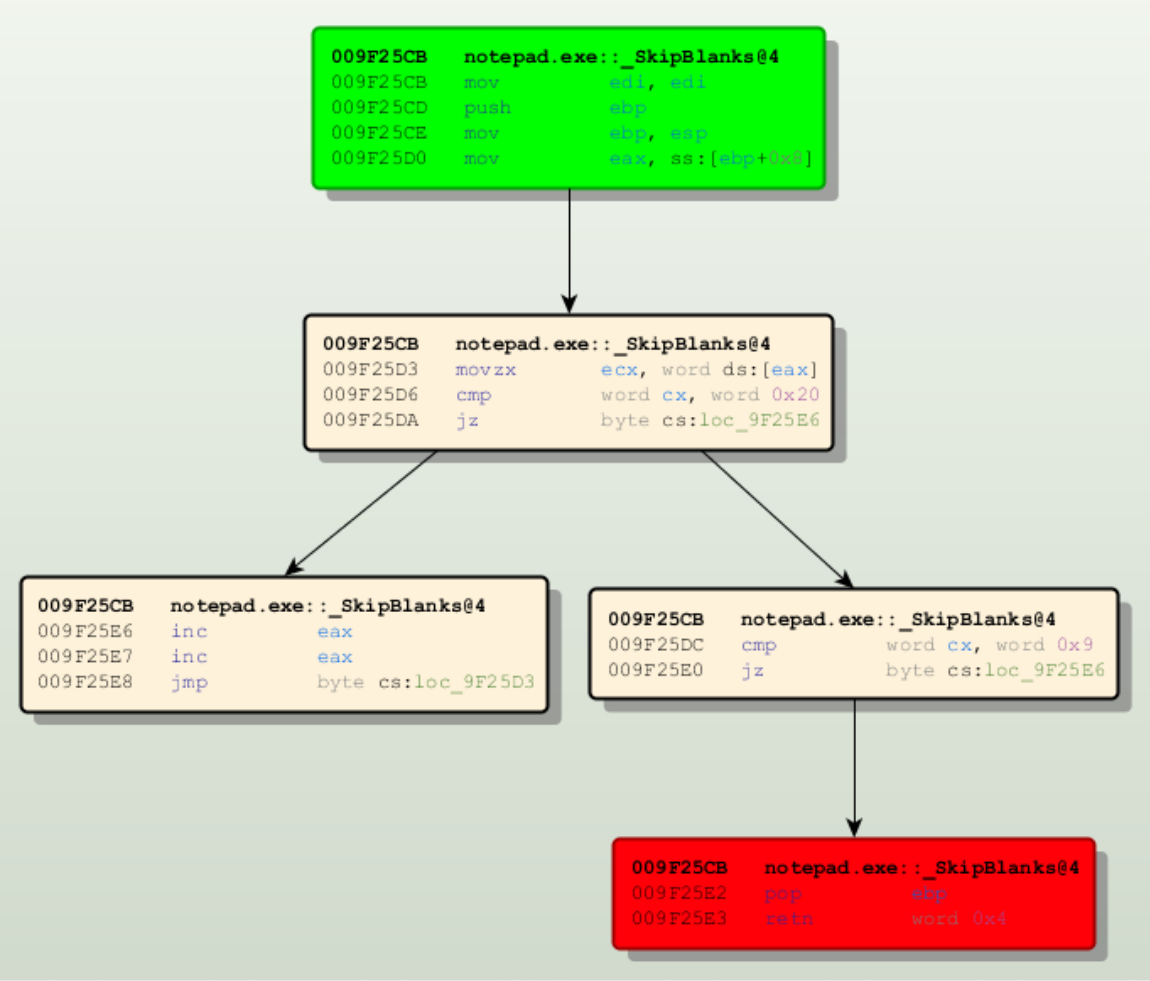
Dominance relation

- Relation about the nodes of a control flow graph.
- "Node A dominates Node B if ***every path*** from the entry node to B includes A".
- Representation: **A *dom* B**
- Properties:
  - Antisymmetric (either *A dom B* or *B dom A*)
  - Reflexive (*A dom A*)
  - Transitive (If *A dom B* and *B dom C* then *A dom C*)
- Can be represented by a tree, the Dominator Tree.

Control Flow Graph



Dominator Tree

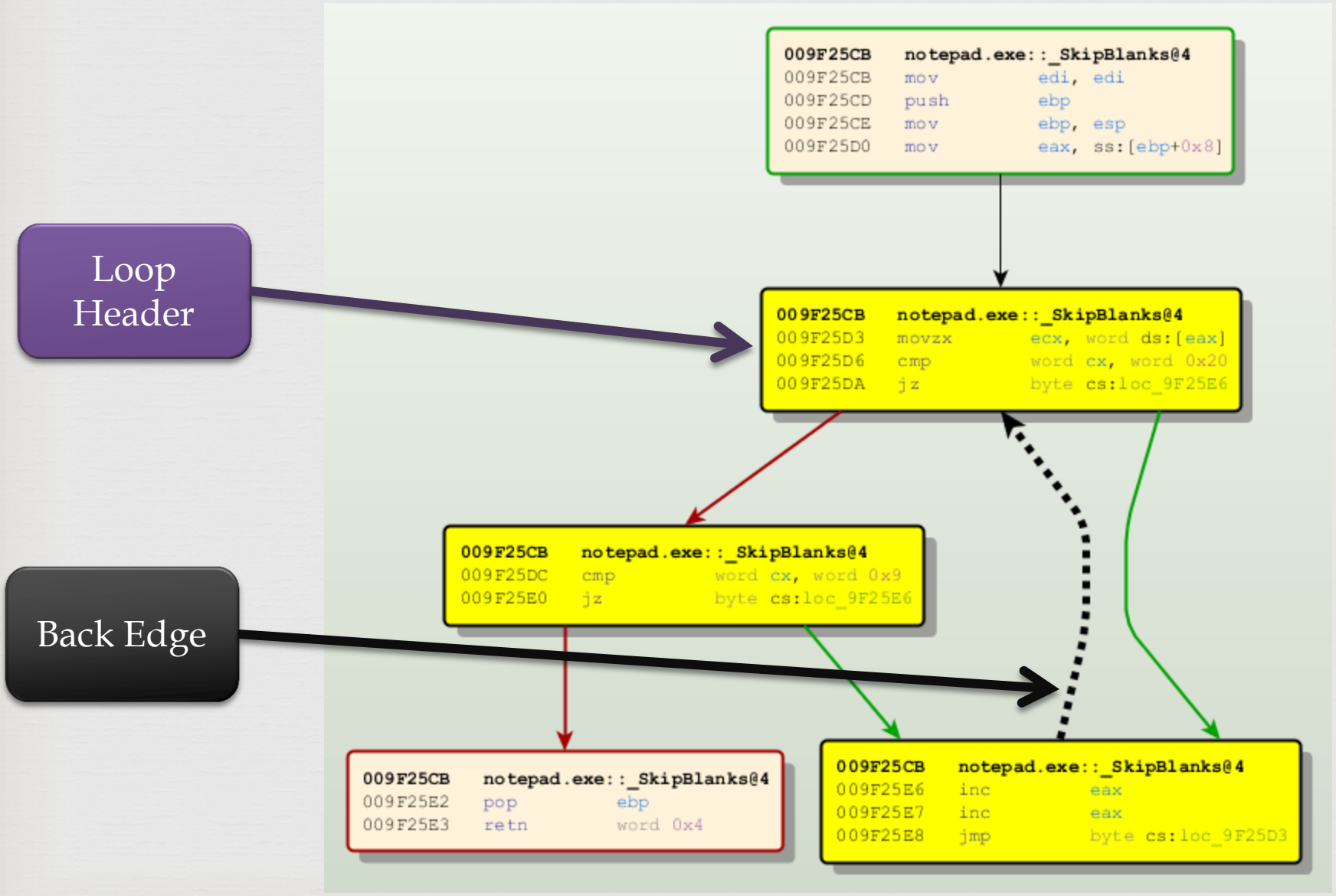


Implementations

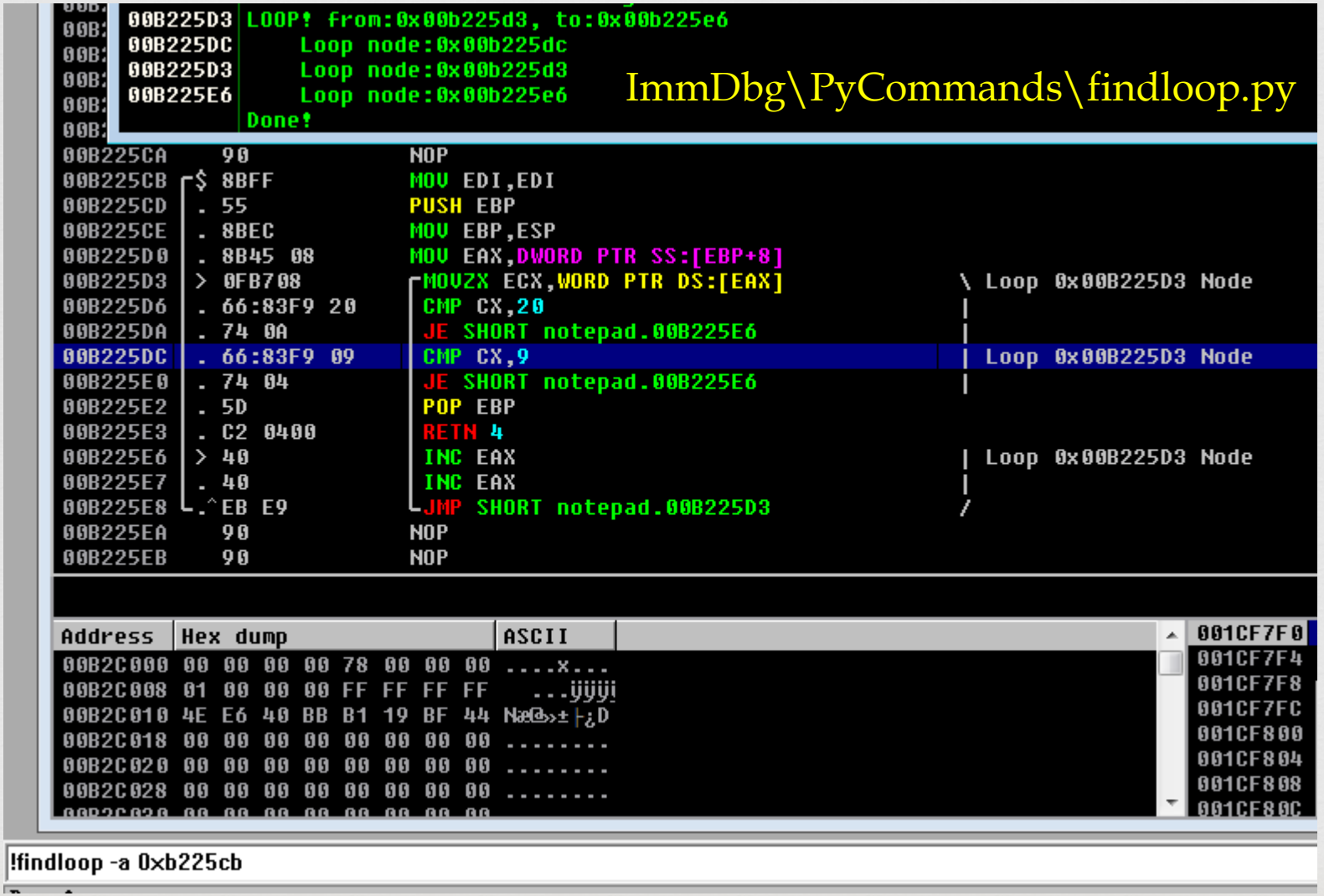
- Classic reference:
  - Lengauer-Tarjan algorithm
- Boost C++ library
- Immunity Debugger
  - libcontrolflow.py
    - Class DominatorTree
- BinNavi API
  - GraphAlgorithms *getDominatorTree()*
  - DEMO: Gui plugin

Natural loops

- We can use the Dominator Tree to identify loops.
- Locate the back edges
- Back edge:
  - An edge whose **head** dominates its **tail**.
- A loop consists:
  - of all nodes dominated by its entry node (head of the back edge) from which the entry node can be reached
- These loops are named *Natural Loops*.


Loop Header
Back Edge
009F25CB notepad.exe::_SkipBlanks@4
009F25CB mov edi, edi
009F25CD push ebp
009F25CE mov ebp, esp
009F25D0 mov eax, ss:[ebp+0x8]
009F25CB notepad.exe::_SkipBlanks@4
009F25D3 movzx ecx, word ds:[eax]
009F25D6 cmp word cx, word 0x20
009F25DA jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25DC cmp word cx, word 0x9
009F25E0 jz byte cs:loc_9F25E6
009F25CB notepad.exe::_SkipBlanks@4
009F25E2 pop ebp
009F25E3 retn word 0x4
009F25CB notepad.exe::_SkipBlanks@4
009F25E6 inc eax
009F25E7 inc eax
009F25E8 jmp byte cs:loc_9F25D3

ImmunityDbg !findloop



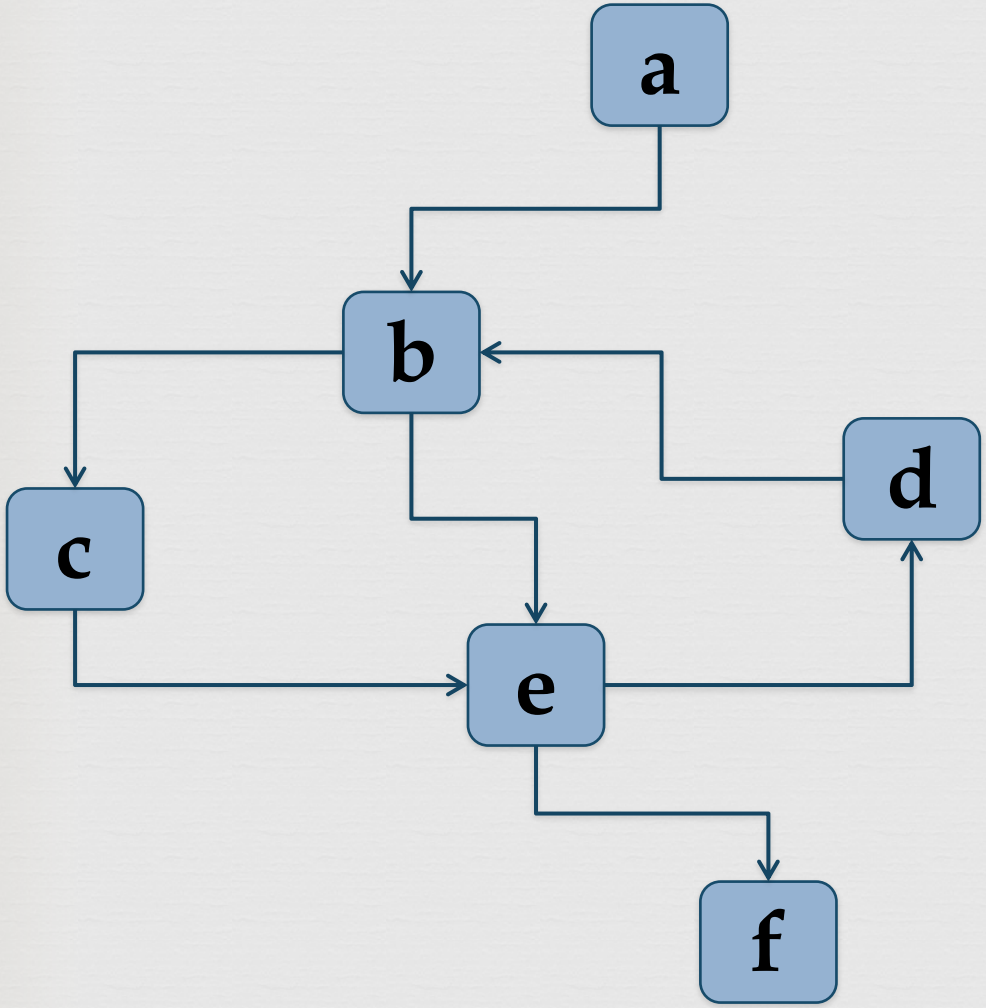
Strongly connected components

Control Flow Analysis

SCC

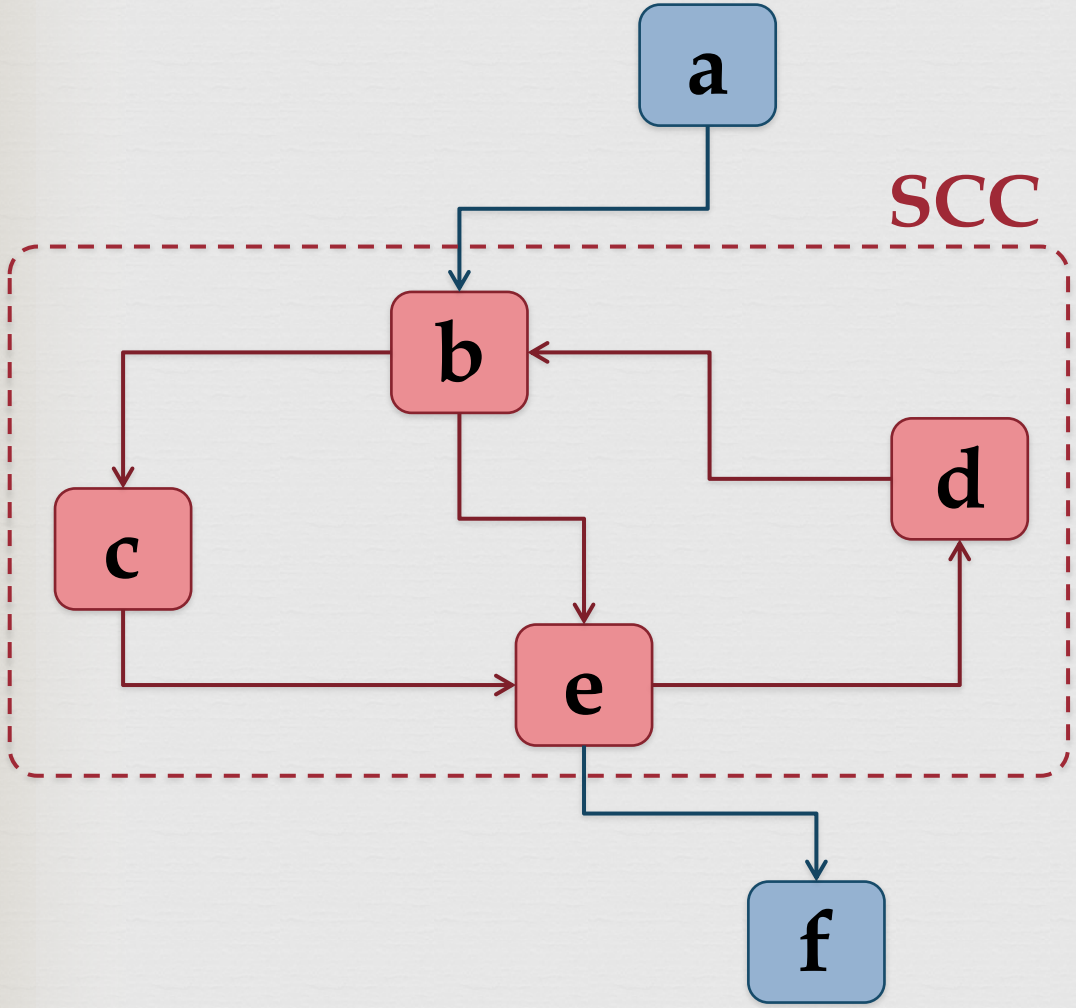
- SCC → Strongly connected components
- A graph (directed/undirected) is called *strongly connected* if there is a path from each *vertex* to every other vertex
- Any loop is a strongly connected component

SCC



This graph is not strongly connected.

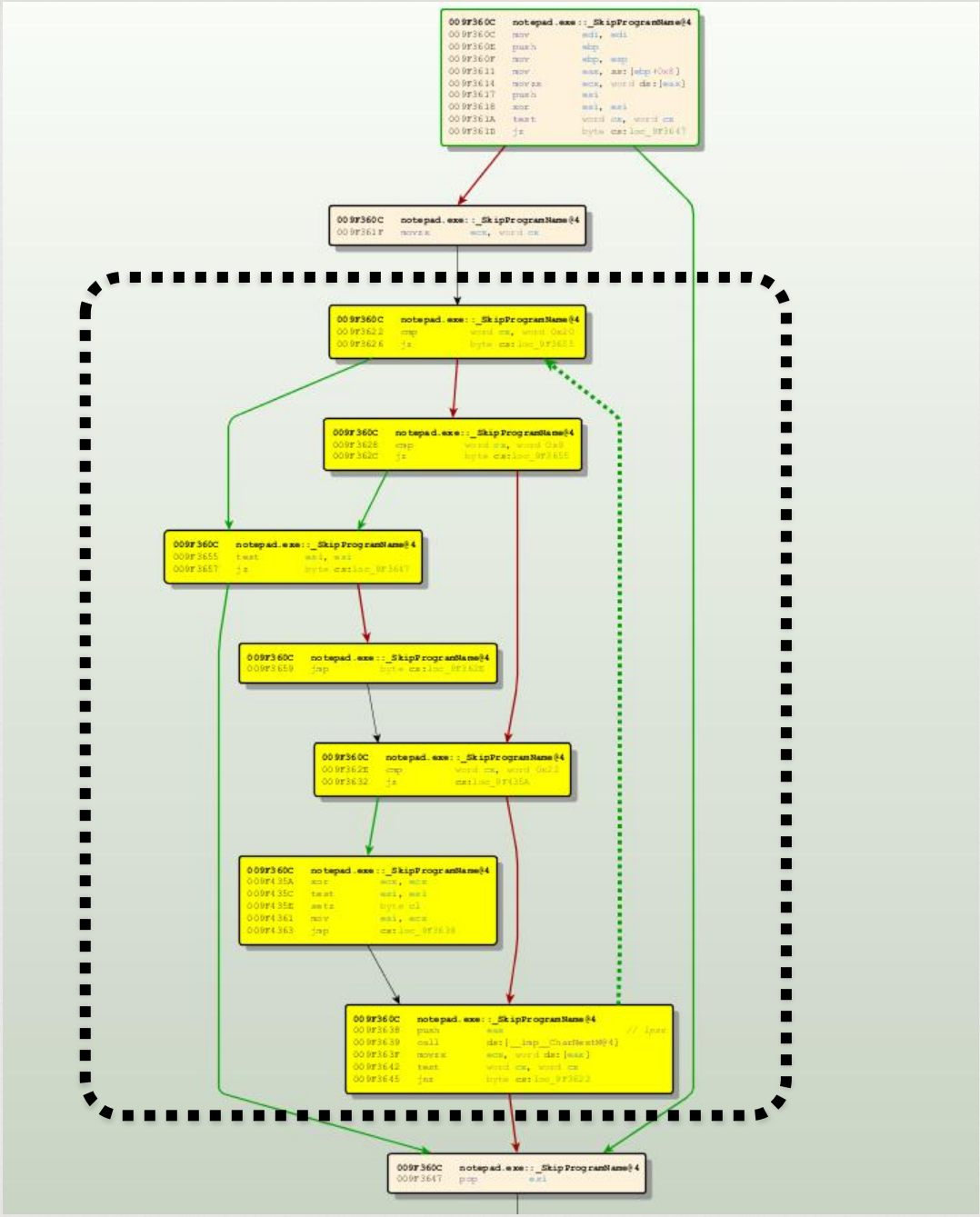
SCC



But it contains a subgraph which is strongly-connected.

SCC - algorithms

- Tarjan algorithm
  - fast algorithm - complex
- Kosaraju-Sharir algorithm
  - simple, but slower than Tarjan's algorithm
- Implementations available for all languages:
  - C#/Python/Lua/Ruby/Java


009F360C notepad.exe::_SkipProgramName@4
009F360C mov edi, edi
009F360E push ebp
009F360F mov ebp, esp
009F3611 mov eax, dword ptr [ebp+0x8]
009F3614 movzx ecx, word ptr ds:[eax]
009F3617 push esi
009F3618 xor esi, esi
009F361A test word ptr ecx, word ptr cx
009F361D jz byte ptr cs:loc_9F3647
009F360C notepad.exe::_SkipProgramName@4
009F361F movzx ecx, word ptr cx
009F360C notepad.exe::_SkipProgramName@4
009F3622 cmp word ptr cx, word 0x20
009F3626 jz byte ptr cs:loc_9F3655
009F360C notepad.exe::_SkipProgramName@4
009F3628 cmp word ptr cx, word 0x9
009F362C jz byte ptr cs:loc_9F3655
009F360C notepad.exe::_SkipProgramName@4
009F3655 test esi, esi
009F3657 jz byte ptr cs:loc_9F3647
009F360C notepad.exe::_SkipProgramName@4
009F3659 jmp byte ptr cs:loc_9F365E
009F360C notepad.exe::_SkipProgramName@4
009F362E cmp word ptr cx, word 0x22
009F3632 jz cs:loc_9F435A
009F360C notepad.exe::_SkipProgramName@4
009F435A xor ecx, ecx
009F435C test esi, esi
009F435E setz byte ptr cl
009F4361 mov esi, ecx
009F4363 jmp cs:loc_9F3638
009F360C notepad.exe::_SkipProgramName@4
009F3638 push eax // lpsz
009F3639 call dword ptr ds:[<&__imp__CharNextW@4>]
009F363F movzx ecx, word ptr ds:[eax]
009F3642 test word ptr cx, word ptr cx
009F3645 jnz byte ptr cs:loc_9F3622
009F360C notepad.exe::_SkipProgramName@4
009F3647 pop esi

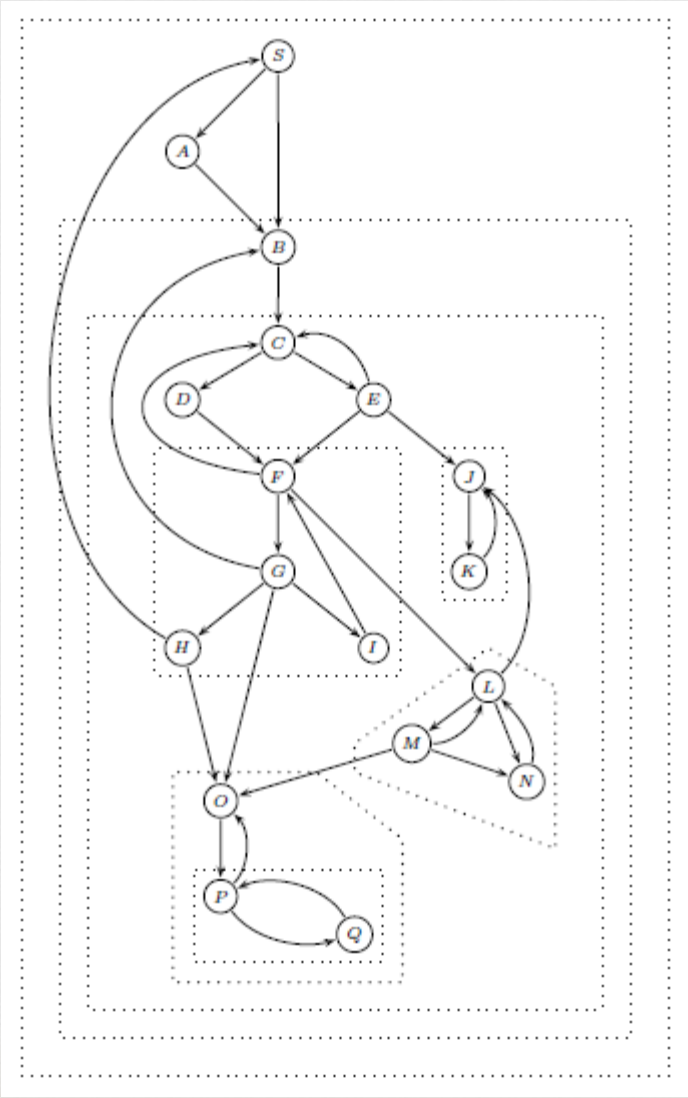
Interval Analysis

Control Flow Analysis

Regions and intervals

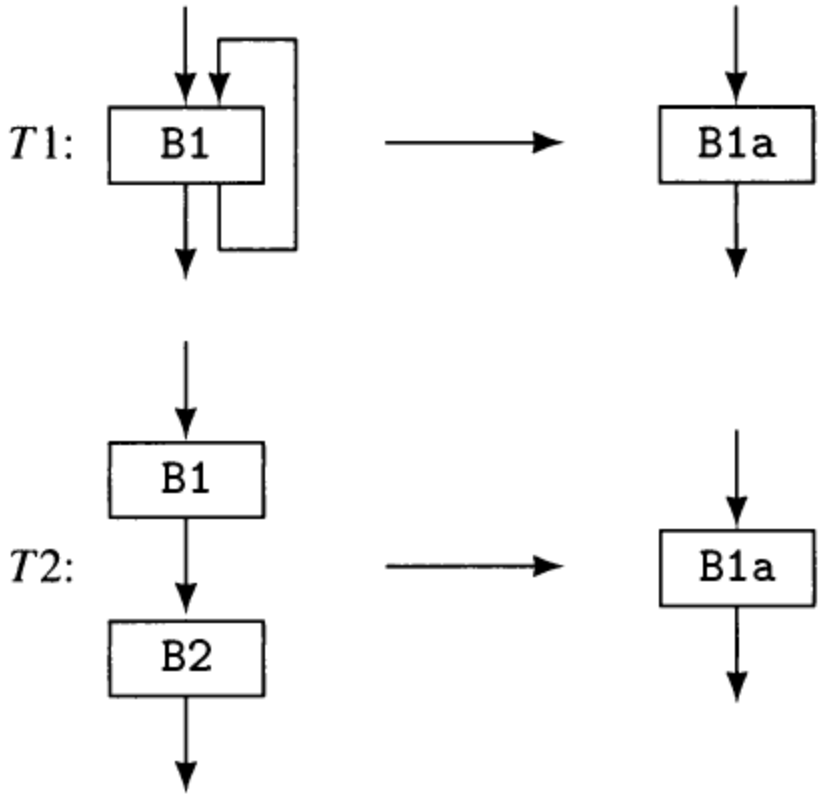
- Unfortunately SCC isn't able to identify *nested* loops
- Interval Analysis
  - Divides the CFG into regions and consolidate them into new nodes (*abstract nodes*) resulting in an *abstract flowgraph.*
- We need to identify *regions* and *pre-intervals*
- Region:
  - A *region* in a flow graph is a sub graph *H* with an unique entry node *h*
- Pre-Interval:
  - A *pre-interval* in a flow graph is a region <*H,h*> such that every cycle (loop) in *H* includes the header *h*.
- Similar to a unique entry SCC.

Nested Intervals

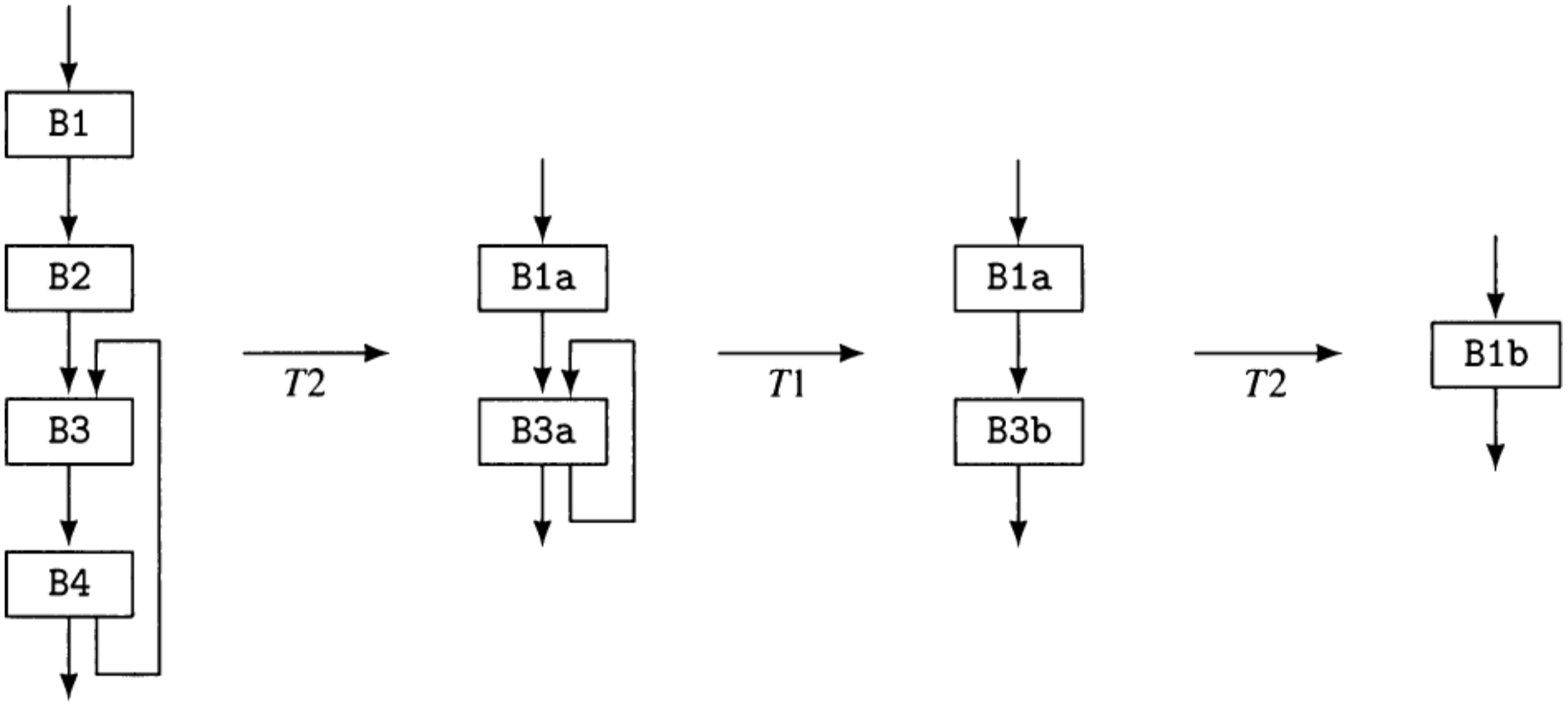


T1/T2 transformations

- Reduction of graphs
- We can collapse nodes from a region to a single node. This is called t1/t2 transformation. If we apply it to all loops, the graph becomes a *cycle-free* one.
- Cycle-free graphs are easier to analyze.



$T1$-$T2$ transformations.

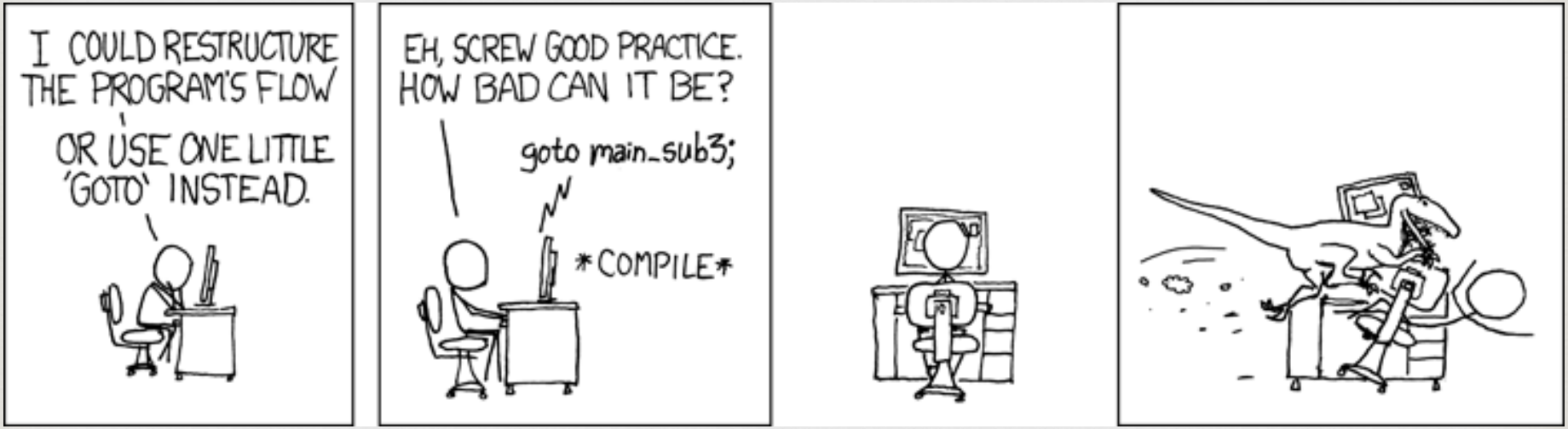


Example of $T1$-$T2$ transformations.

Interval analysis

- DEMO

GOTO considered harmful…

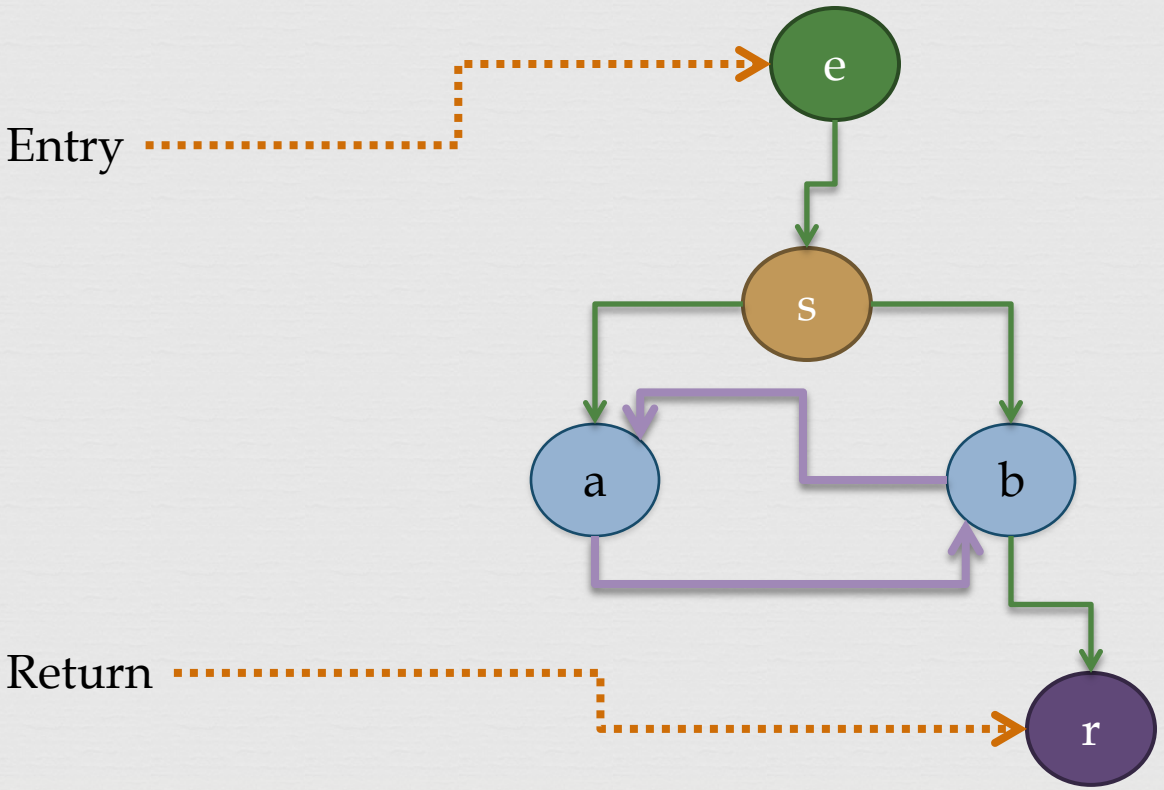


http://xkcd.com/292/

Irreducible graphs

- All the loops identified by the previous methods (dominance tree/interval analysis) are called *natural loops*.
- They are *unique entry* loops.
- There another type of loop:
  - irreducible graphs or improper regions

Irreducible graph



Loop (a , b)

2 entries! b or a

Irreducible graphs

- Who codes like that?
  - Anyone who uses GOTO
  - It is rare, but it does exist
    - notepad.exe
    - ntoskrnl.exe (Windows Kernel)
- What's the problem?
  - Most of the algorithms are unable to handle irreducible graphs!!! Including Interval analysis.
  - Can't apply T1/T2

translateString

```
int *__stdcall TranslateString(int a1)
{
        wchar_t v1; // cx@1
        …
        if ( v1 )
        {
                while ( 1 )
                {
                        v5 = &v22 + v26;
                        …
                        LABEL_49:
                        v1 = *(_WORD *)v7;
                                …
                }
        }
        goto LABEL_49;
}
```

Jump inside the WHILE statement

Solutions

- There are 2 main solutions to handle *irreducible graphs*:
  - Structural Analysis
  - DJ-Graphs

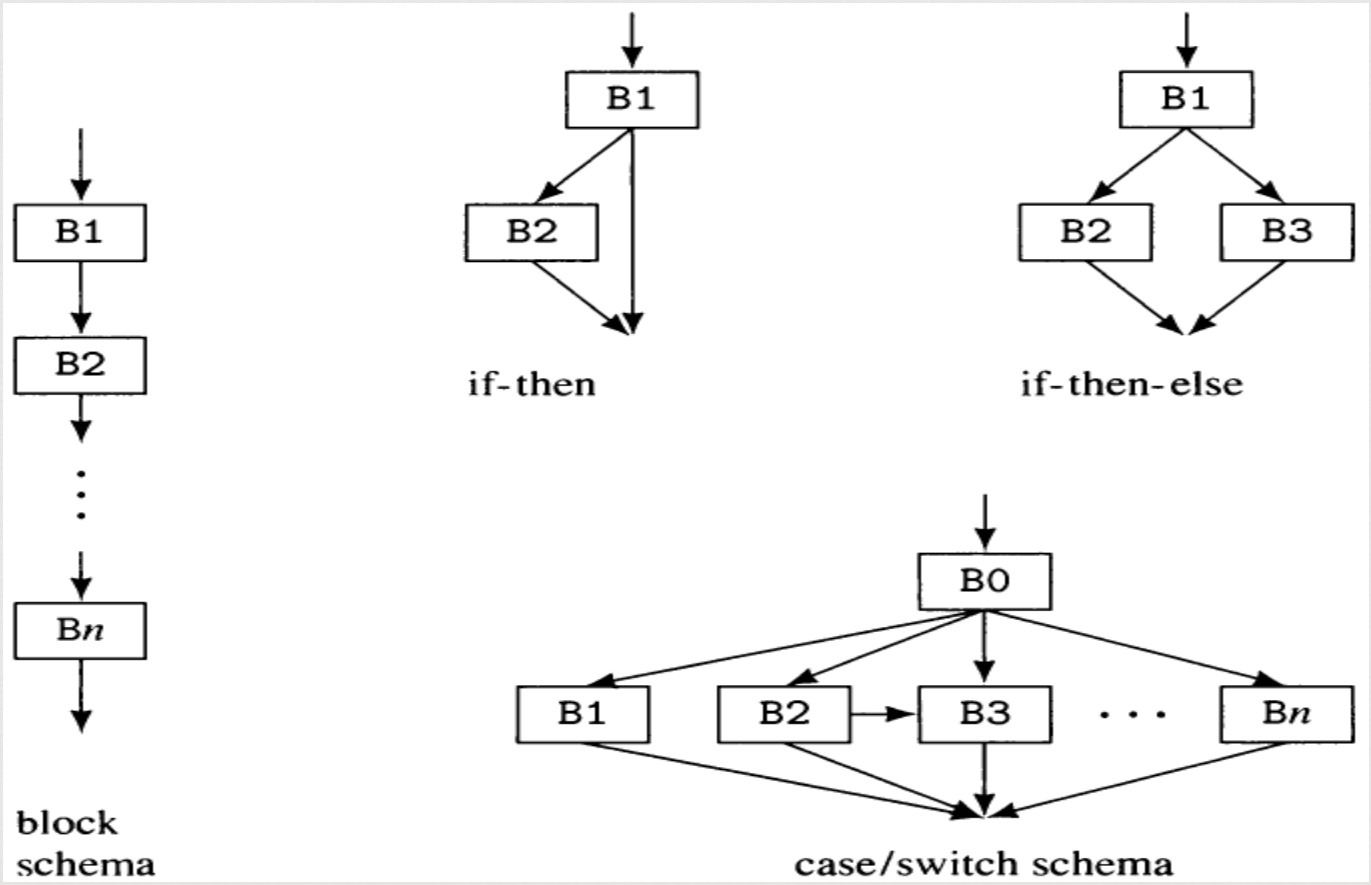
Structural Analysis

Control Flow Analysis

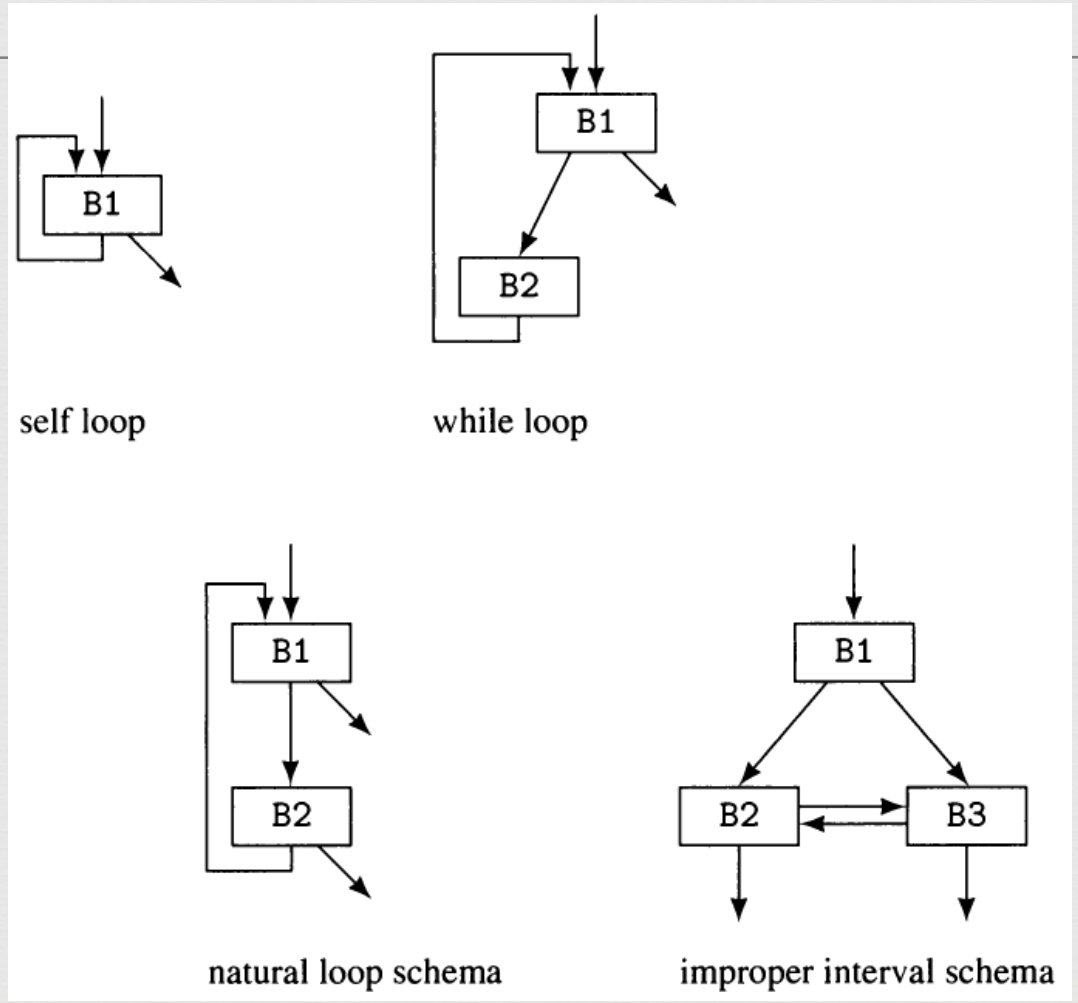
Structural Analysis

- Structural analysis will identify the main language constructs inside a flow graph using *region schemas*.
- Do you want to build your own decompiler?
  - Hex-Rays decompiler internally uses Structural Analysis
- Created by Micha Sharir
- Reference paper:
  - Structural analysis: a new approach to flow analysis in optimizing compliers (1979)

Acyclic schemas



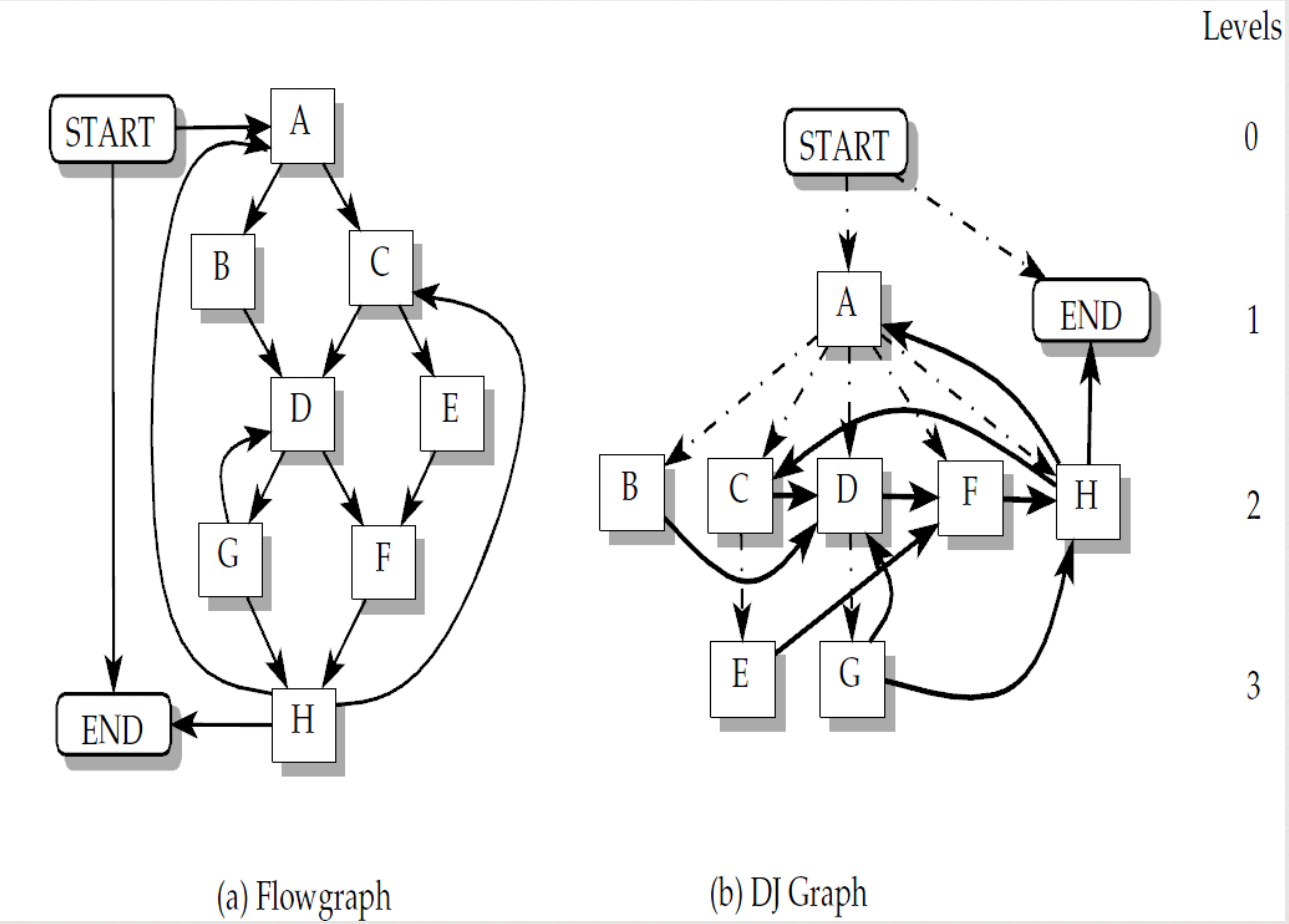
Cyclic schemas



DJ-Graphs

- Another way to handle *irreducible graphs*.
- It is also able to identify all types of structures, including improper regions and nested structures.
- Uses a combination of the dominance tree and the original flowgraph with two additional types of edges:
  - the D edge (D*ominator*)
  - the J edges
- Paper: *Identifying loops using DJ graphs*.[e]

DJ-Graphs


START
A
B
C
D
E
G
F
H
END
(a) Flowgraph
Levels
0
1
2
3
START
A
END
B
C
D
F
H
E
G
(b) DJ Graph

Applications

- Taint analysis
  - Control dependency (dominators, post-dominators)
- Diff Slicing
  - Execution Indexing (view the CFG as a grammar)
    - Execution alignment
  - Identification of root causes of software crashes
- Decompilation
- Code coverage
- Bug finding

References

- a - http://www.usenix.org/event/usenix03/tech/full_papers/prasad/prasad_html/node5.html
- b - An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries
- c – Bertrand Anckaert, Matias Madou, and Koen De Bosschere. 2006. A model for self-modifying code. In *Proceedings of the 8th international conference on Information hiding* (IH'06)
- d - http://www.jakstab.org/
- e - Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1996. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst.* 18, 6 (November 1996), 649-658.
- f - Advanced compiler implementation – Steven Muchnick
- g - Notes on Graph Algorithms Used in Optimizing Compilers - Carl D. Offner

Questions?

- Contact: *edgarmb@gmail.com*
  edgar@research.coseinc.com
- twitter: ***@embarbosa***