Automating the identification of data structures inside binaries

Edgar Barbosa

SyScan 2012

Singapore

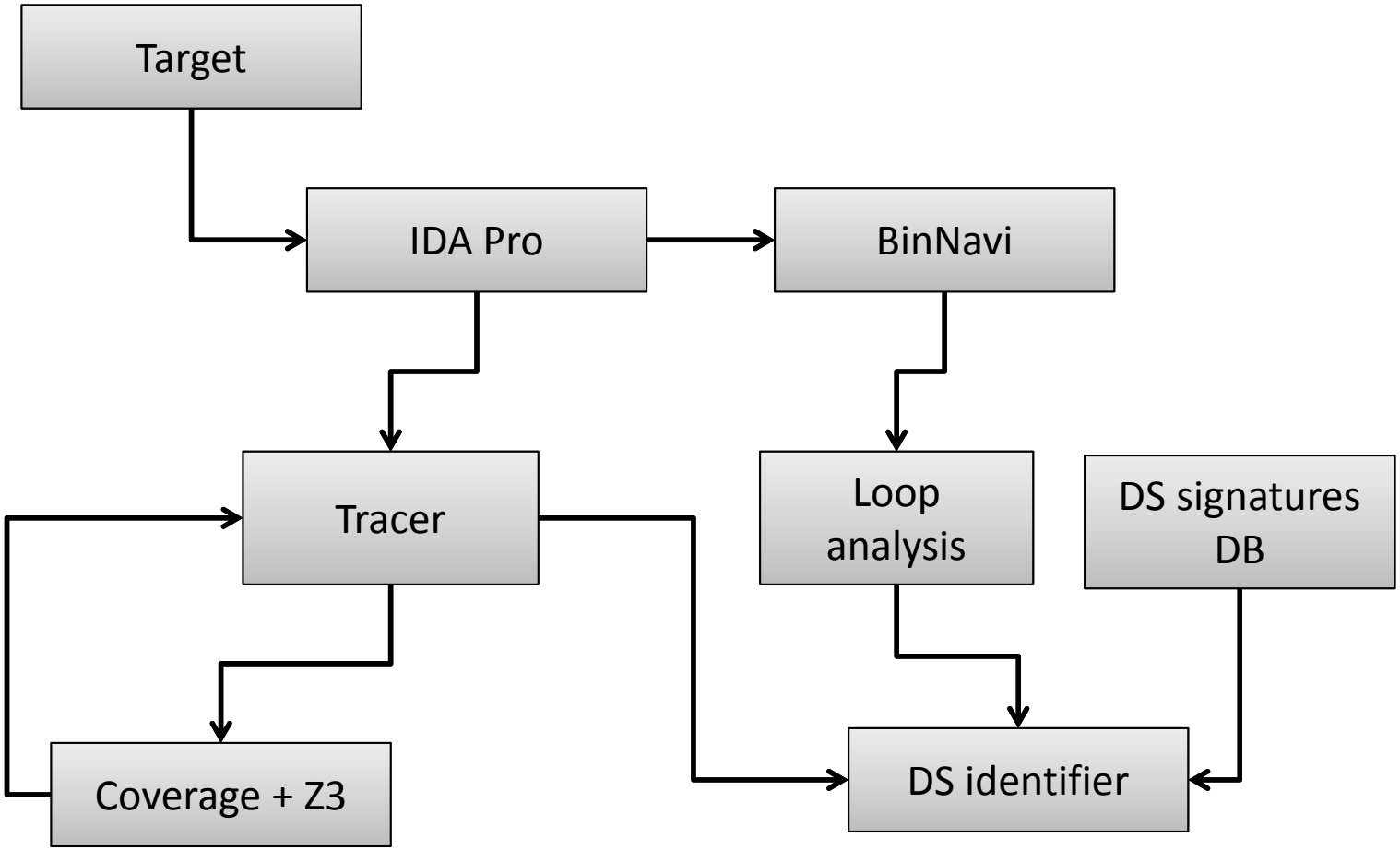
Info

- Edgar Barbosa
- Senior Security Researcher – COSEINC
- Virtualization, Windows Internals, Rootkits, Program Analysis and SMT solvers.
- aka opc0de
- http://twitter.com/embarbosa
- edgarmb@gmail.com

Objective

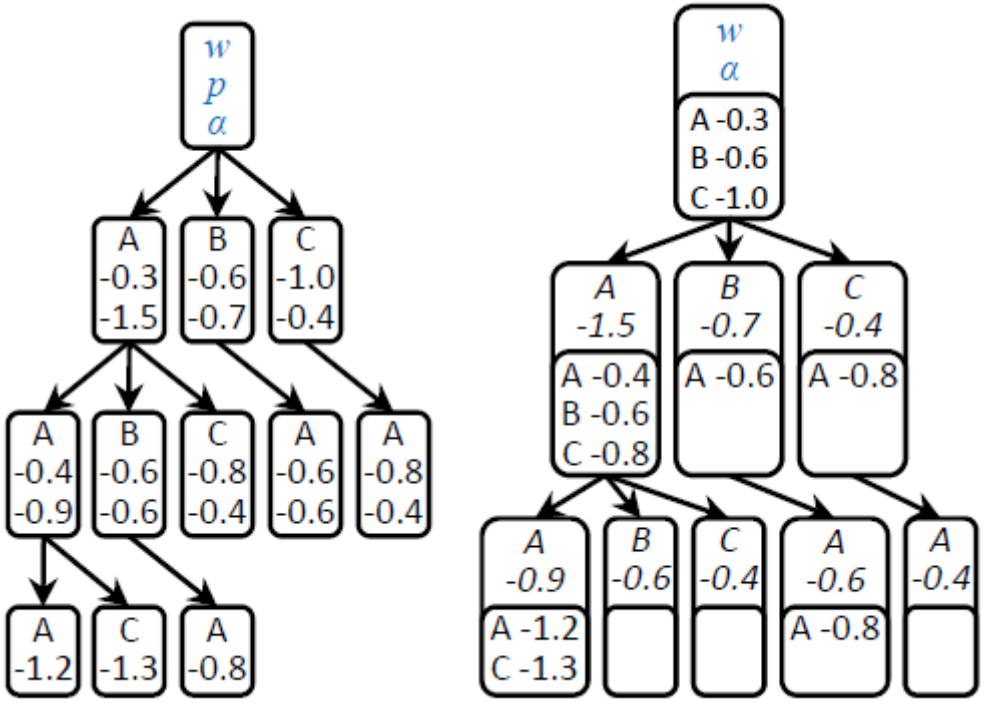
- To present several techniques and heuristics to automate the **identification** of data structures.
- Tools:
  - Computational linguistics (NLP)
    - n-grams and skip-grams
  - Static analysis
  - Dynamic analysis
  - Constraint satisfaction (SMT solver)

System Architecture



Assumptions

- No access to PDB data structure information (types)
- Operating System/Compiler agnostic
- x86 architecture - 32-bits
- No type propagation.



SyScan'12

N-GRAMS

http://groups.csail.mit.edu/sls/publications/2008/Hsu_Interspeech08.pdf

n-grams - definition

- “In the field of computational linguistics and probability, an **n-gram** is a contiguous **sequence** of **n** items from a given sequence of *text* or *speech*” - Wikipedia
- The items in question can be *phonemes*, *syllables*, *letters*, *words* or application specific *terms*.

n-grams

- Example
  - DNA sequencing
    - AGCTTCGA
  - n = 1 (unigram)
    - {A, G, C, T, T, C, G, A}
  - n = 2 (bigram)
    - {AG, GC, CT, TT, TC, CG, GA}
  - n = 3 (trigram)
    - {AGC, GCT, CTT, TTC, TCG, CGA}

n-grams

- Applications:
  - statistical natural language processing (NLP) – (Google books n-gram viewer)
  - data compression
  - machine translation - (Google Translate)
  - malware detection and classification
    - text = sequence of instructions
    - or sequence of function calls

skip-grams

- Similar to n-gram but allows the items to be “skipped”.
- Applications:
  - string matching
    - Longest common subsequence (LCS)
  - information retrieval systems (IR)

skip-grams

instruction sequence

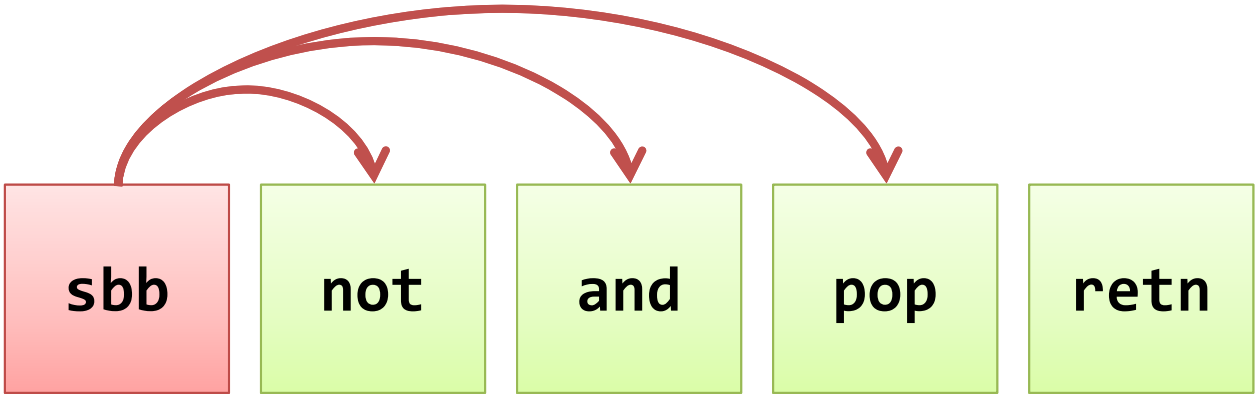
```
sbb     eax, eax
not     eax
and     eax, edx
pop     ebp
retn    4
```

opcode sequence

```
sbb
not
and
pop
retn
```

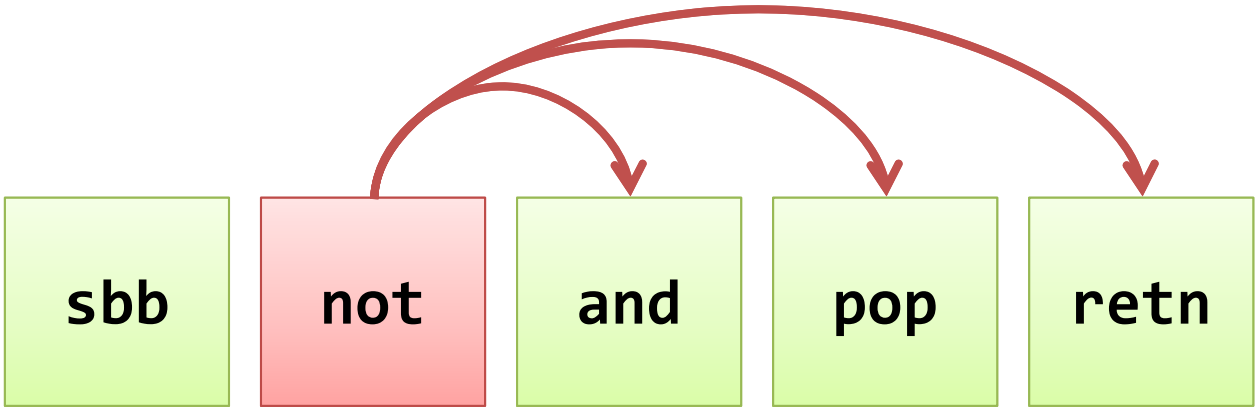
We will extract the 2-skip-gram set of the opcode sequence

2-skip-bigrams



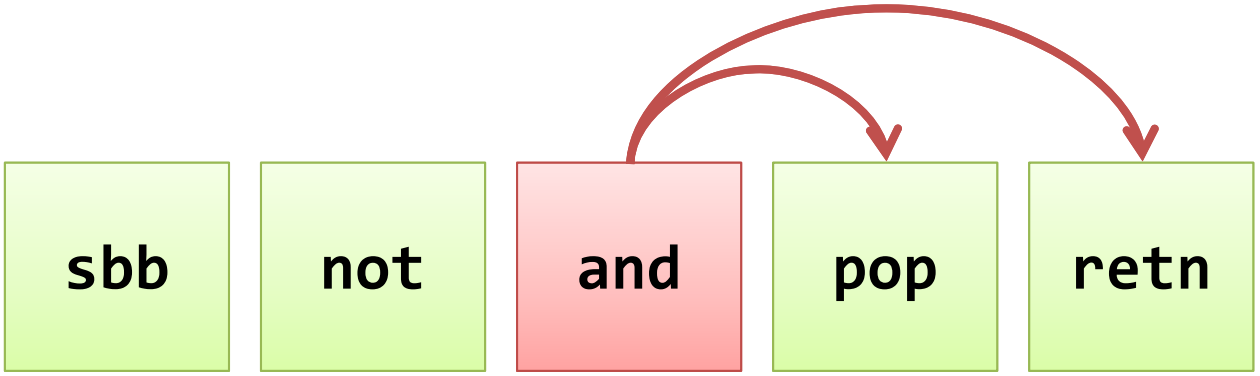
{sbb not, sbb and, sbb pop}

2-skip-bigrams



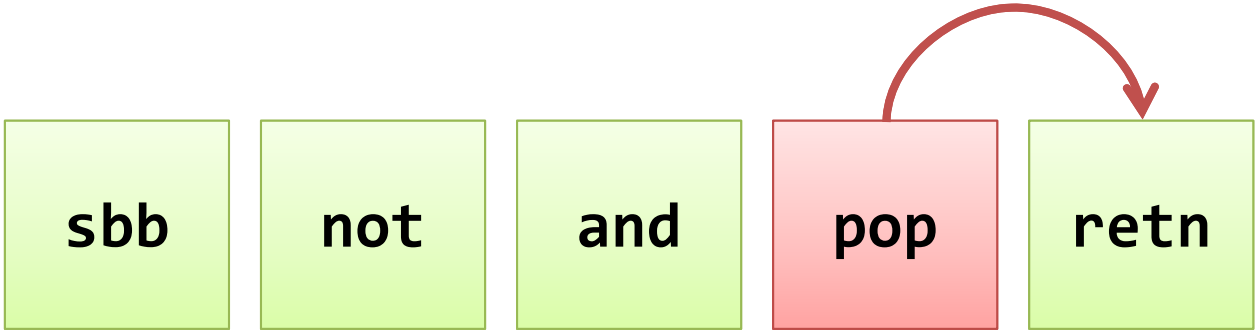
{not and, not pop, not retn}

2-skip-bigrams



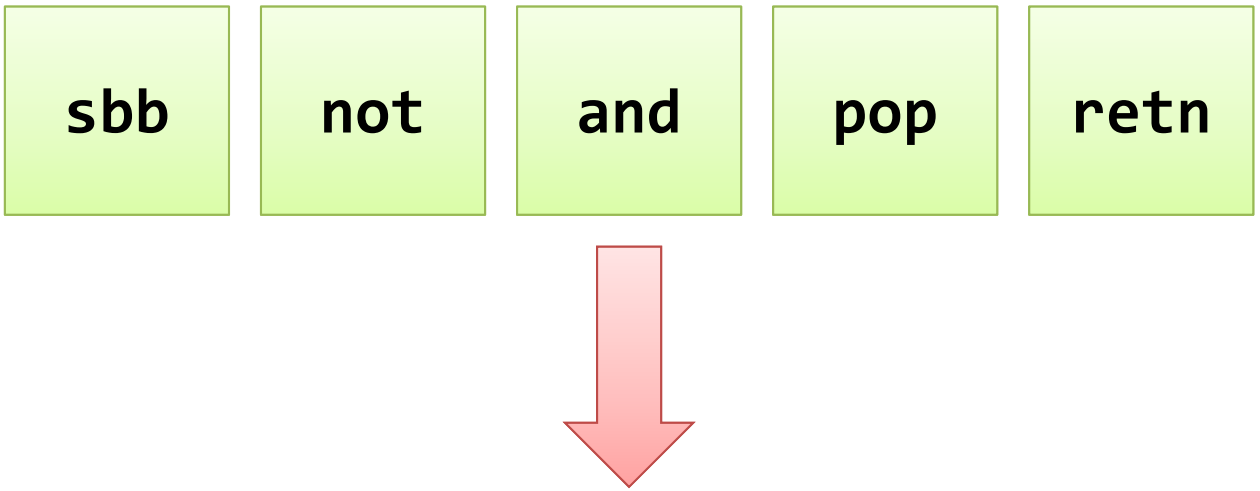
{and pop, and retn}

2-skip-bigrams



{pop retn}

2-skip-bigrams – Set



{sbb not, sbb and, sbb pop,
not and, not pop, not retn,
and pop, and retn, pop retn}

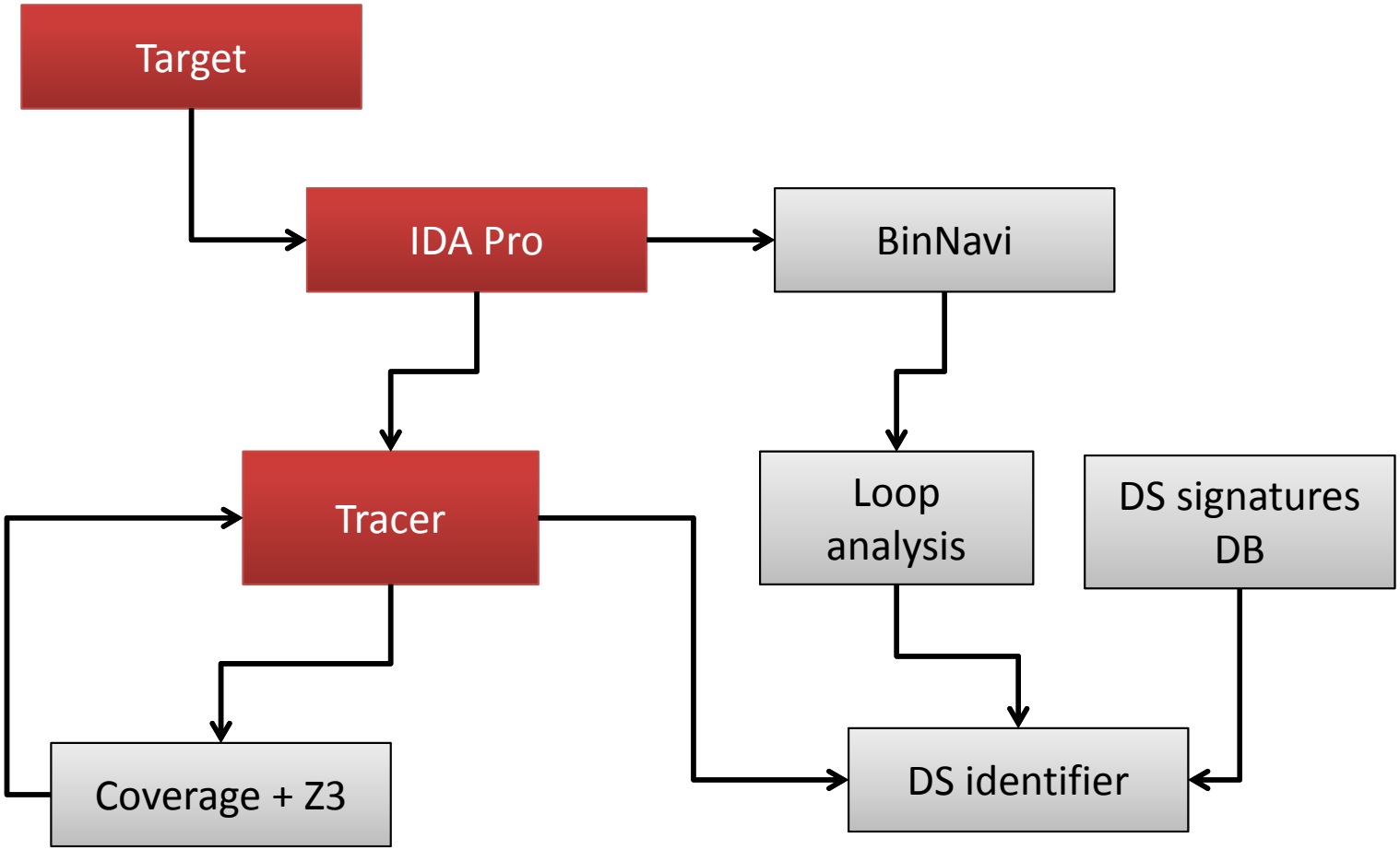
skip-grams and data structures

- The idea is to **generate** a big collection of n-grams/skip-grams from **large sequences** of computer instructions and **correlate** them to data structure **patterns/signatures**.
- We use dynamic and static analysis tools to generate the instruction sequences and then **pattern matching** over the n-gram/skip-grams to **identify** data structures.

SyScan'12

DYNAMIC ANALYSIS

Architecture



Objective

- Generate a sequence of executed instructions.
- Extract skip-grams of the sequence (trace file)
- Example:
  - Trace function F until it returns
  - Function:
    - HeapCreate (ntdll.dll)
  - Save the trace to a trace file.
  - Generate trace events.

Tracer

- Execution tracer (IDAPython script)
- Select the start function and then trace-until-ret
- Trace file contents:
  - Instruction (address, raw bytes, disassembly)
  - Memory access
    - Address, type (r/w), size
  - Context (regs)
  - Eflags

Tracer events

- The architecture is based on events. The clients can subscribe for any of following events:
  - trace-start / trace-end
  - instr-exec
  - memory-read / memory-write
  - function-call / function-return
  - stack-access (local / parameters)
  - branch events (conditional/unconditional jmps)
  - system calls (sysenter)

events

```
>> mov [ebp-0x4], edi
[{'address': '0x77702e23L',
  'disasm': 'mov [ebp-0x4], edi',
  'event-name': 'instr-exec',
  'opcode': 'mov'},
 {'event-name': 'mexp-eval',
  'info': {'base': 'ebp',
           'evaluated': '0x1af9f0L',
           'expression': '[ebp-0x4]',
           'immediate': '-0x4',
           'index': '-',
           'scale': '-',
           'selector': '-',
           'type': 'indirect'}},
 {'event-name': 'mem-write', 'size': '4', 'source': 'edi',
'target': '#eval'}]
```

HeapCreate trace

- Trace-until-ret of

  HeapCreate(0x0, 0x1000, 0x4000)

  function
- 4600 inst-exec events
- 3 syscall events

HeapCreate tracefile (slice)

- ('0x77702af4L', 'CMP [EBP+0x14], EDI', '397d14')
  [PRE-REGS] eax:0x00010000 ebx:0x00000004 ecx:0x00004000
  edx:0xffffffff ebp:0x001dfb10 esp:0x001dfa54 edi:0x00000000
  esi:0x00001000 efl:0x00000206
  [0=EVAL]:0x001dfb24[VALUE:0x1000]

- ('0x77702af7L', 'JZ 0x7771d558', '0f845baa0100')
  [PRE-REGS] eax:0x00010000 ebx:0x00000004 ecx:0x00004000
  edx:0xffffffff ebp:0x001dfb10 esp:0x001dfa54 edi:0x00000000
  esi:0x00001000 efl:0x00000206

- ('0x77702afdL', 'MOV EBX, [EBP-0x1c]', '8b5de4')
  [PRE-REGS] eax:0x00010000 ebx:0x00000004 ecx:0x00004000
  edx:0xffffffff ebp:0x001dfb10 esp:0x001dfa54 edi:0x00000000
  esi:0x00001000 efl:0x00000206
  [1=EVAL]:0x001dfaf4[VALUE:0x1f0000]

tracefile entry

- instr-exec address, disasm, raw bytes
  - ('0x77702af4L', '**CMP [EBP+0x14]**, EDI', '397d14')
- context – (before execution)
  - `[PRE-REGS] eax:0x00010000 ebx:0x00000004 ecx:0x00004000 edx:0xffffffff ebp:0x001dfb10 esp:0x001dfa54 edi:0x00000000 esi:0x00001000 efl:0x00000206`
- [EBP+0x14] is a memory expression. EVALuate the expression and read the memory value
  - `[0=EVAL]:0x001dfb24[VALUE:0x1000]`

skip-gram (variation)

- We want to extract meaningful skip-grams from the executed instructions.
- Most of the important skip-grams are those where the instructions are linked by a **data dependency** relationship.
- We are basically creating skip-grams where the skip value is based on data dependency.

Data dependency - trace

```
mov     eax, [edx+8]
inc     ebx
dec     ecx
mov     edx, eax
add     edi, 0x8
```

Data dependency

```
mov     eax, [edx+8]
inc     ebx
dec     ecx
mov     edx, eax
add     edi, 0x8
```

Data dependency

- Result: (bigram)

```
{‘mov eax, [edx+8]’,
 ‘mov edx, eax’}
```

- On the second instruction:
  - **edx** depends on the value of eax *and*
  - **eax** value depends of the value of [edx+8]

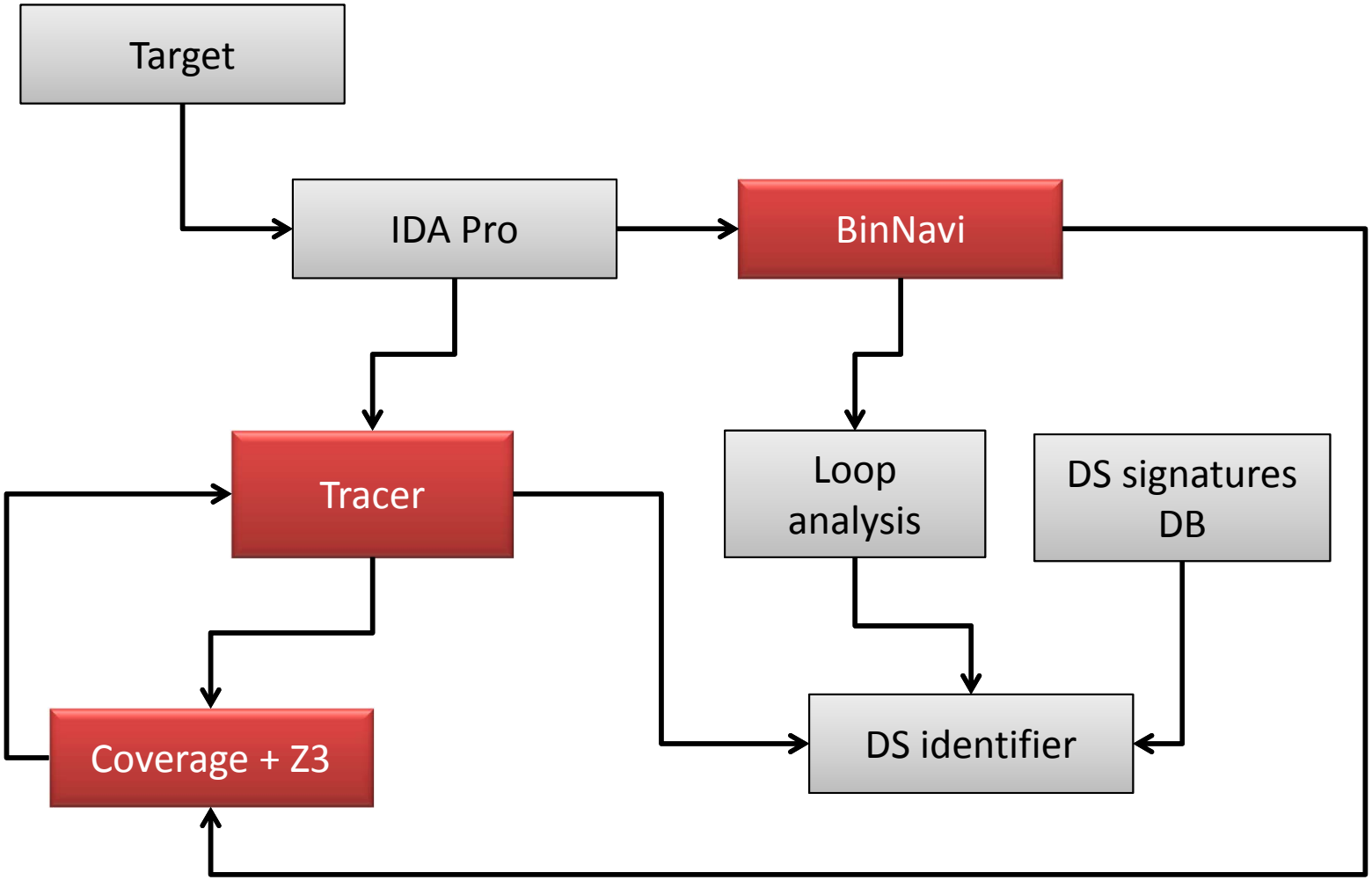
Dynamic analysis

- We now have the tracefile with detailed information about each instr-exec event.
- We generated skip-grams of the trace. Some skip-grams are based on data dependency analysis.
- It is necessary to collect the maximum amount of n-grams from the traces to increase the accuracy of the system. In dynamic analysis, this means we need the maximum **code coverage.**

```
304                         ImageStack s = imagePlus.getStack();
305                         switch(imageType) {
306                         case ImagePlus.GRAY8:
307                         case ImagePlus.COLOR_256:
308                                 slices_data_b = new byte[depth][];
309                                 for( int z = 0; z < depth; ++z )
310                                         slices_data_b[z] = (byte []) s.getPixels( z + 1 );
311                                 break;
312                         case ImagePlus.GRAY16:
313                                 slices_data_s = new short[depth][];
314                                 for( int z = 0; z < depth; ++z )
315                                         slices_data_s[z] = (short []) s.getPixels( z + 1 );
316                                 break;
317                         case ImagePlus.GRAY32:
318                                 slices_data_f = new float[depth][];
319                                 for( int z = 0; z < depth; ++z )
320                                         slices_data_f[z] = (float []) s.getPixels( z + 1 );
321                                 break;
```


COVERAGE

Architecture



Constraint solving and code coverage

- We are using SMT solvers to increase code coverage.
- SMT solver: Microsoft Z3
- Basic process:
  - Translation of x86 instructions to REIL
  - Constraint extraction
  - Constraint solving
  - Generate new input values

IL

- What about Klee?
  - Klee is a very powerful tool for generating input to increase code coverage, but it depends on the LLVM intermediate language.
- Ours projects uses REIL.
- An Intermediate Language created by Zynamics (acquired by Google)
  - REIL Access via BinNavi API (Python)

REIL instruction set

- No side effects
- 17 instructions
- Very easy to create new analysis on the top of REIL.

REIL- arithmetic instructions

- `add` – addition of 2 values
- `sub` – subtraction of 2 values
- `mul` – unsigned multiplication
- `div` – unsigned division
- `mod` – unsigned module
- `bsh` – logical shift operation

REIL – bitwise instructions

- `and` – Boolean and
- `or` – Boolean or
- `xor` – Boolean exclusive or

REIL – data transfer instructions

- ldm – load a value from memory
- stm – store a value to memory
- str – store a value in a register

REIL - conditional

- BISZ – compare a value to zero
  - set output operand to 1 if value is zero
- JCC – conditional jump

```
000000010025D60D   bisz  word t10,  , byte ZF
000000010025DA00   jcc   byte ZF,  , 0x10025E6
```

REIL - others

- undef
- unkn
- nop

REIL - support

- General purpose x86 instructions
- Doesn’t support:
  - FPU
  - sse, sse2, sse3, mmx
  - System instructions (LGDT,…)
  - Segment override prefixes – (Windows TEB/PEB)
    - FS:[0x18] becomes [0x18] !

Translation (x86 → Reil)

- x86 basic block:

```
010025D3   movzx        ecx, word ds:[eax]
010025D6   cmp          word cx, word 0x20
010025DA   jz           loc_10025E6
```

Translation (x86 → Reil)

```
000000010025D300: ldm [DWORD eax, EMPTY , WORD t0]
000000010025D301: or [DWORD 0, WORD t0, DWORD ecx]
000000010025D600: and [DWORD ecx, WORD 65535, WORD t1]
000000010025D601: and [WORD t1, WORD 32768, WORD t2]
000000010025D602: and [WORD 32, WORD 32768, WORD t3]
000000010025D603: sub [WORD t1, WORD 32, DWORD t4]
000000010025D604: and [DWORD t4, DWORD 32768, WORD t5]
000000010025D605: bsh [WORD t5, WORD -15, BYTE SF]
000000010025D606: xor [WORD t2, WORD t3, WORD t6]
000000010025D607: xor [WORD t2, WORD t5, WORD t7]
000000010025D608: and [WORD t6, WORD t7, WORD t8]
000000010025D609: bsh [WORD t8, WORD -15, BYTE OF]
000000010025D60A: and [DWORD t4, DWORD 65536, DWORD t9]
000000010025D60B: bsh [DWORD t9, DWORD -16, BYTE CF]
000000010025D60C: and [DWORD t4, DWORD 65535, WORD t10]
000000010025D60D: bisz [WORD t10, EMPTY , BYTE ZF]
000000010025DA00: jcc [BYTE ZF, EMPTY , DWORD 16786918]
```

Express the x86 instruction semantics with no side effects

Microsoft Z3 – what is it?

- “Z3 is a state-of-the art **theorem prover** from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. Z3 offers a compelling match for **software analysis** and verification tools, since **several common software constructs map directly into supported theories**.” - Microsoft

Solving constraints

- How to ask the Z3 solver questions about constraints?

```
add eax, ebx
sub eax, 0x50
cmp eax, 0x40
jz  _branch2
```

- →Find **eax** and **ebx** values to satisfy the JZ conditional jump.

Solving constraint

- The system now translates the x86 instructions to REIL and generate a **Formula** to be solved by the Z3 solver.
- The formula uses several Z3 commands/operators.
- Most the of formulas uses the BV (BitVector) theory, which is very appropriate for program analysis.

z3 - formula

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))
(declare-const d (_ BitVec 32))
(assert (= c (bvadd a b)))
(assert (= d (bvsub c #x00000050)))
(assert (= d #x00000040))
(check-sat)
(get-model)
```

http://rise4fun.com/z3 - online Z3

z3 - output

```
sat
    (model
        (define-fun d () (_ BitVec 32) #x00000040)
        (define-fun c () (_ BitVec 32) #x00000090)
        (define-fun b () (_ BitVec 32) #x00000000)
        (define-fun a () (_ BitVec 32) #x00000090)
    )
```

- model:

EAX = 0x00000090 (a)

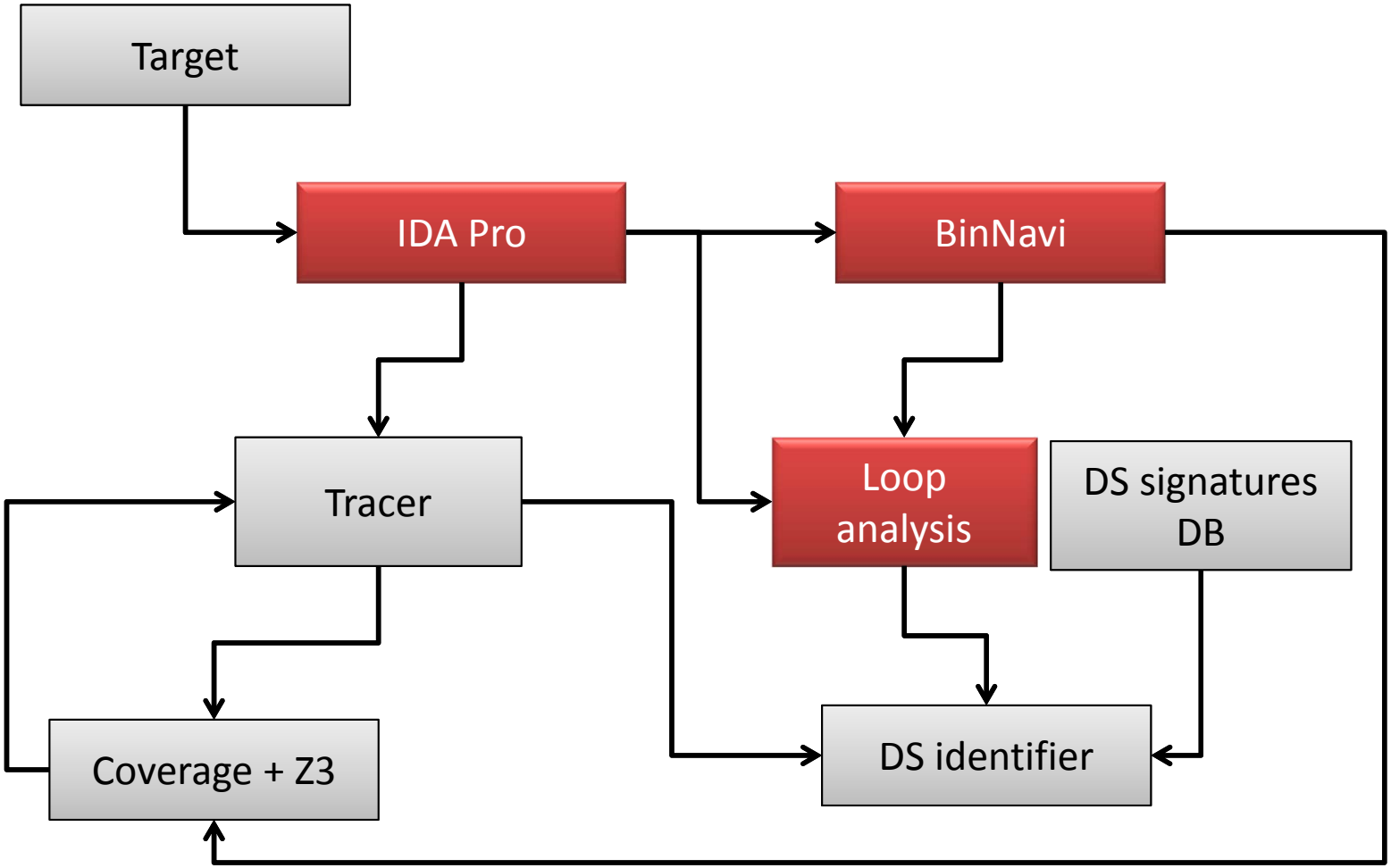
EBX = 0x00000000 (b)

x86 → REIL → z3

- Translation from REIL to Z3 formulas using the bit-vector theory is very straightforward.
- Z3 is a very, very powerful solver and the documentation is very good.
- But do not use it for everything:
  - "Z3 is a low level tool. It is **best used as a component** in the context of **other tools** that require solving logical formulas." – Z3 tutorial

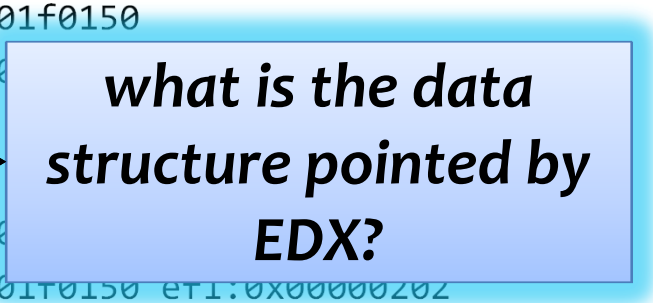
STATIC ANALYSIS

Architecture



trace (HeapCreate) - slice

```
('0x77702643L', 'JB 0x77706b50', '0f8207450000')
eax:0x0000014b ebx:0x001f0000 ecx:0x00000000 edx:0x001f0150
ebp:0x001dfa44 esp:0x001dfa28 edi:0x001f0588 esi:0x001f0150 efl:0x00000202
('0x77702649L', 'MOV ECX, [EDX]', '8b0a')
eax:0x0000014b ebx:0x001f0000 ecx:0x00000000 edx:0x001f0150
ebp:0x001dfa44 esp:0x001dfa28 edi:0x001f0588 esi:0x00
[MEXP-EVAL]:0x001f0150
('0x7770264bL', 'TEST ECX, ECX', '85c9')
eax:0x0000014b ebx:0x001f0000 ecx:0x00000000 edx:0x00
ebp:0x001dfa44 esp:0x001dfa28 edi:0x001f0588 esi:0x001f0150 efl:0x00000202
('0x7770264dL', 'JNZ 0x77706ba5', '0f8552450000')
eax:0x0000014b ebx:0x001f0000 ecx:0x00000000 edx:0x001f0150
ebp:0x001dfa44 esp:0x001dfa28 edi:0x001f0588 esi:0x001f0150 efl:0x00000246
('0x77702653L', 'MOV ECX, [EDX+0x4]', '8b4a04')
[PRE-REGS] eax:0x0000014b ebx:0x001f0000 ecx:0x00000000 edx:0x001f0150
ebp:0x001dfa44 esp:0x001dfa28 edi:0x001f0588 esi:0x001f0150 efl:0x00000246
[MEXP-EVAL]:0x001f0154
```



Static analysis

- No operation is executed on the ECX value because ecx value is NULL and then the JNZ is not satisfied.
- We could find a value for ecx able to satisfy the constraint using Z3.
- But again:
  - Don’t use SMT solvers for everything!
- For some cases we can just apply some static analysis tools

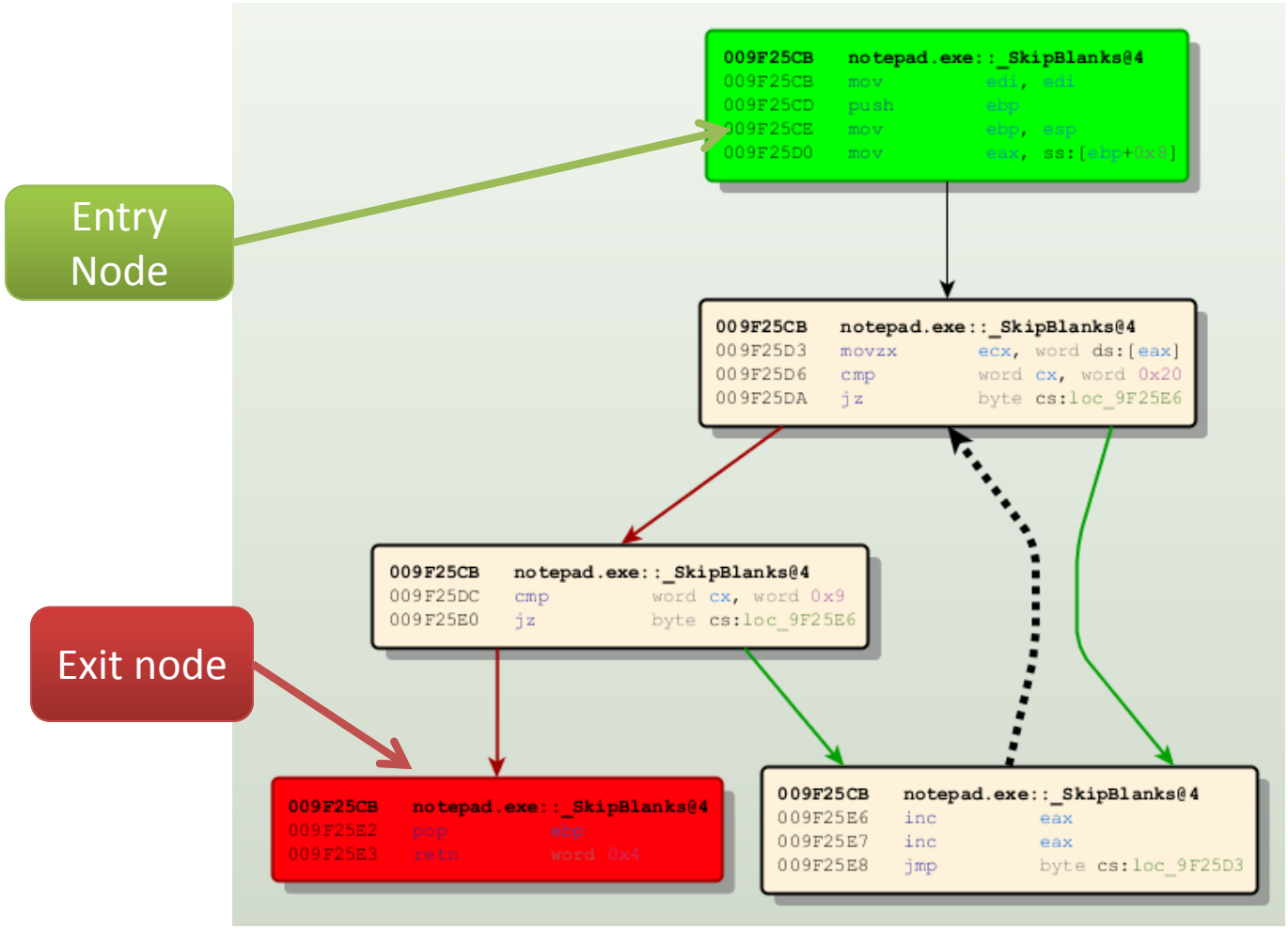
Loop analysis

- One of the best places for recovery of data structure (especially recursive data structures) are the **loop structures**.
- It becomes much easier to detect some data structures once you have identified the loop structures.
- But loop identification is not so easy!

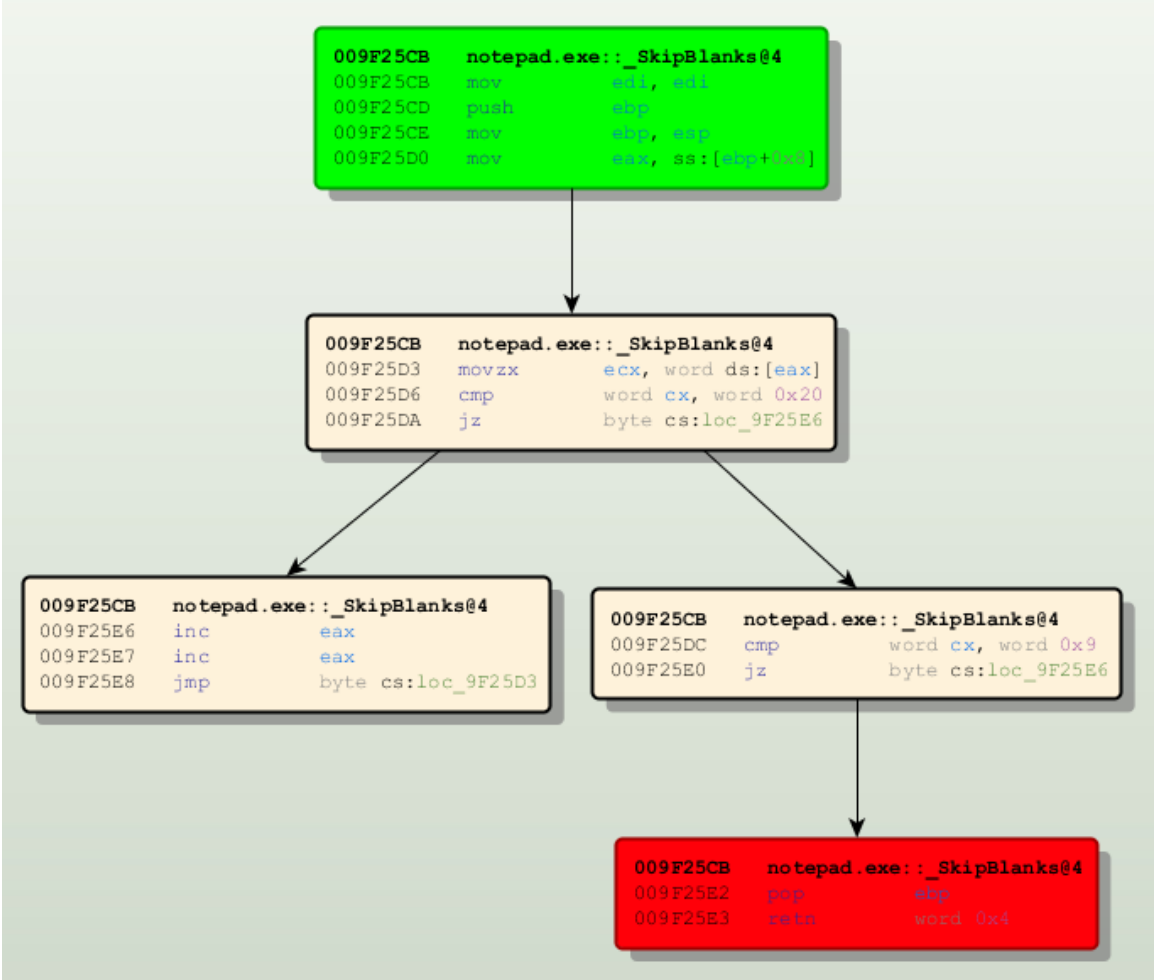
Dominance relation

- First step: Generate the Dominator Tree.
- Dominance: Relation about the nodes of a control flow graph.
- “Node A dominates Node B if ***every path*** from the **entry node of the CFG** to B includes A”.
- Representation: **A *dom* B**
- Can be represented by a tree structure, known as the **Dominator Tree**.

Control Flow Graph



Dominator Tree

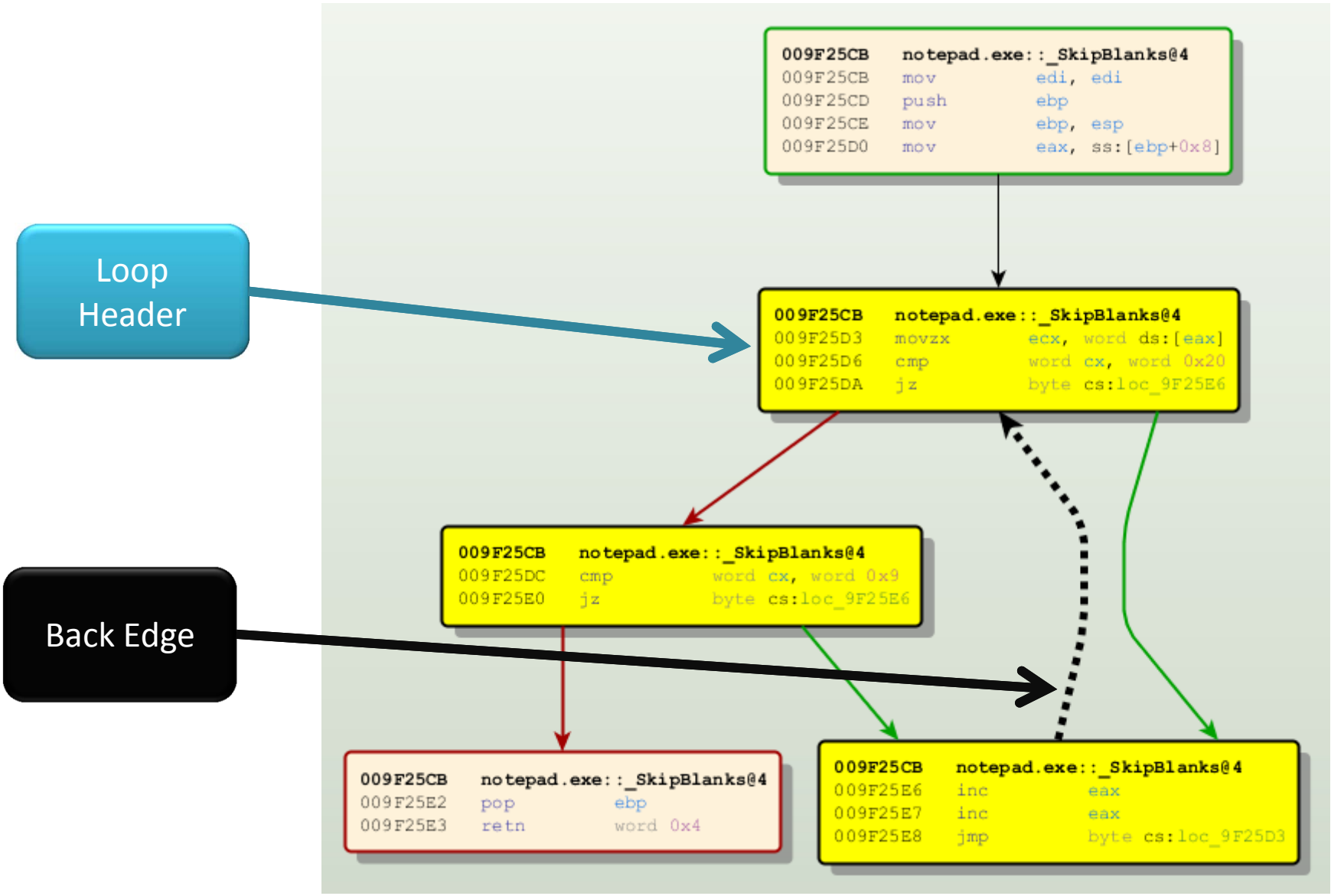


Implementations

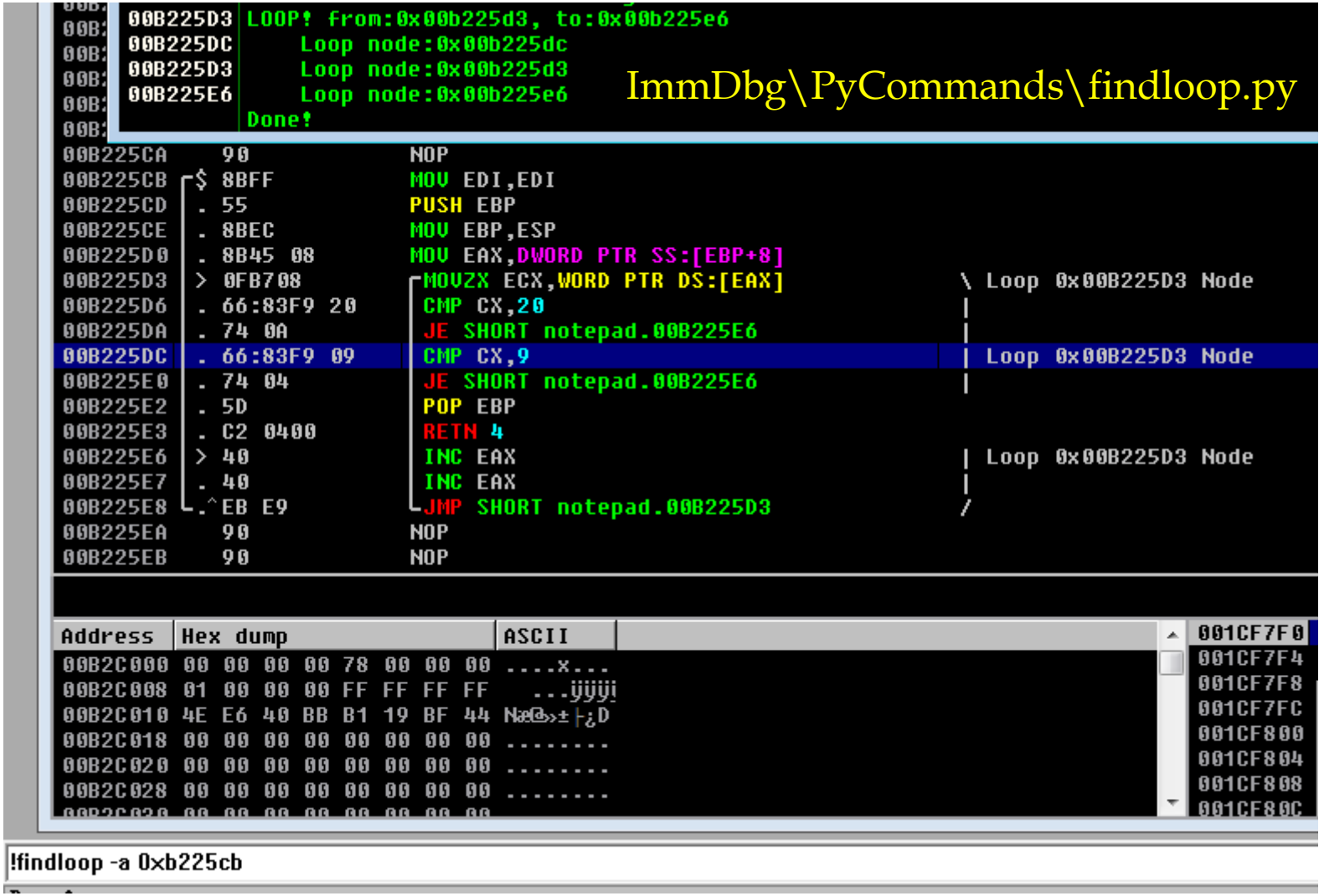
- Classic implementation reference:
  - Lengauer-Tarjan algorithm
- Immunity Debugger
  - lDominatorTree Class inside libcontrolflow.py
- BinNavi API
  - GraphAlgorithms *getDominatorTree()*
  - DEMO: Gui plugin

Natural loops

- We can use the Dominator Tree to identify loops.
- Locate the back edges
- Back edge is an edge where the **head** dominates its **tail**.
- Then a loop consists:
  - **of all nodes dominated by its entry node from which the entry node can be reached** (easier to see in the next illustration)
- The identified 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

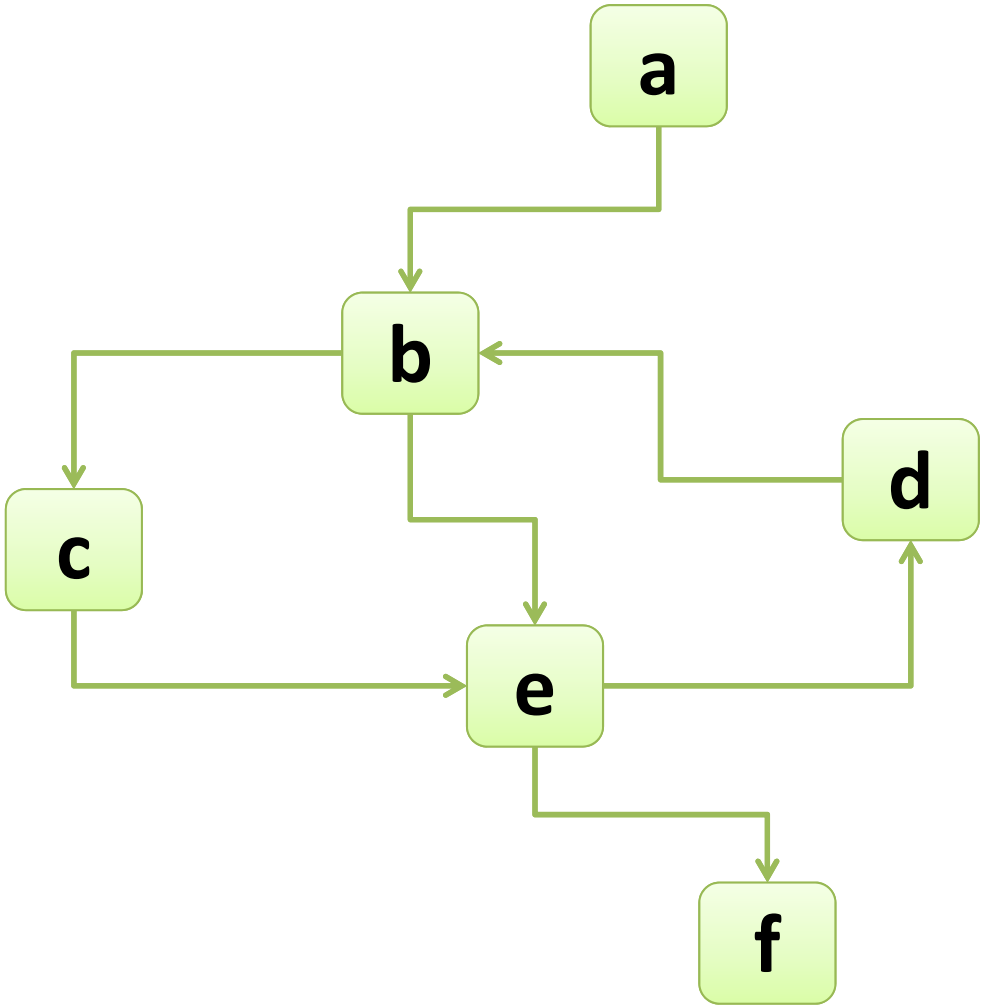
ImmunityDbg !findloop



SCC

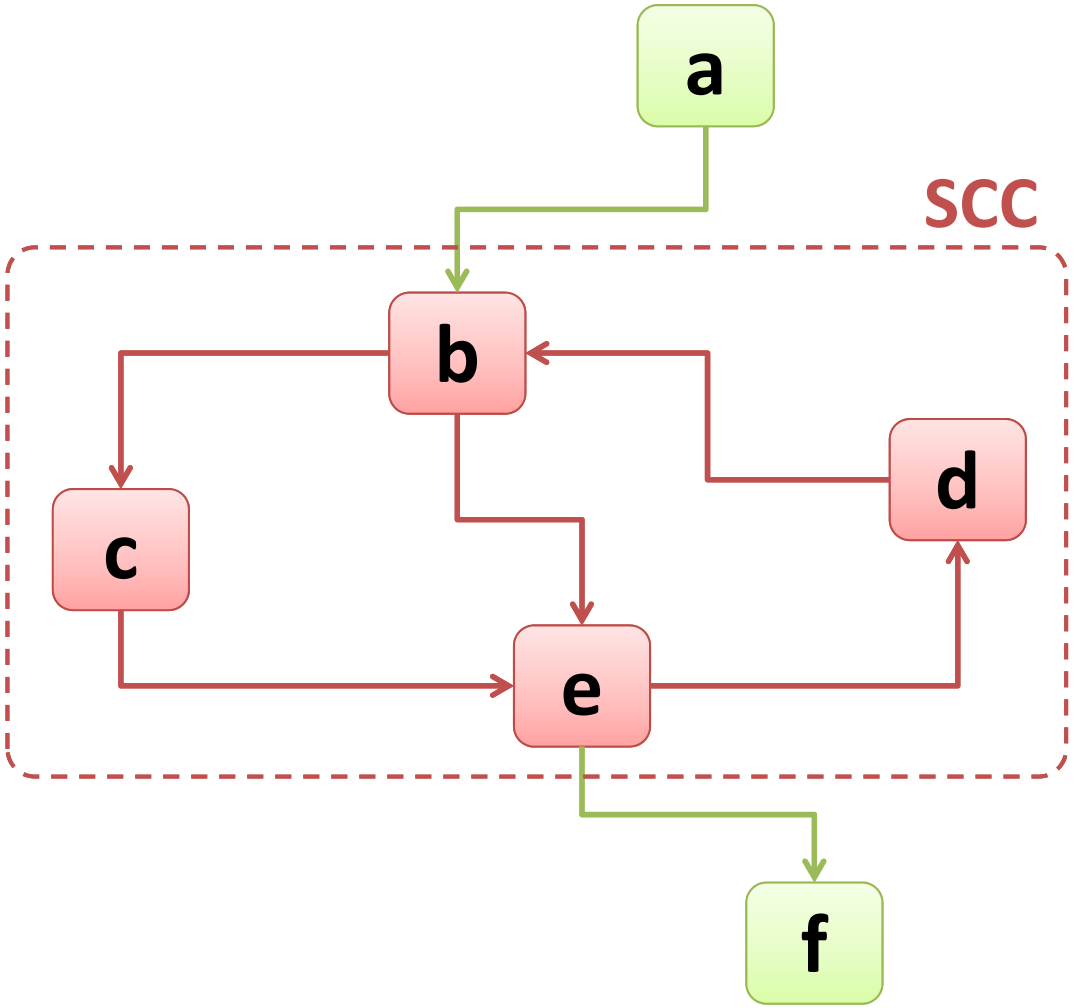
- There is another way to find loop (generic)
- SCC → Strongly connected components
- A graph is called *strongly connected* if there is a path from each *vertex* to **every other** vertex
- **Important:**
  - **All loop structures are strongly connected components!**

SCC



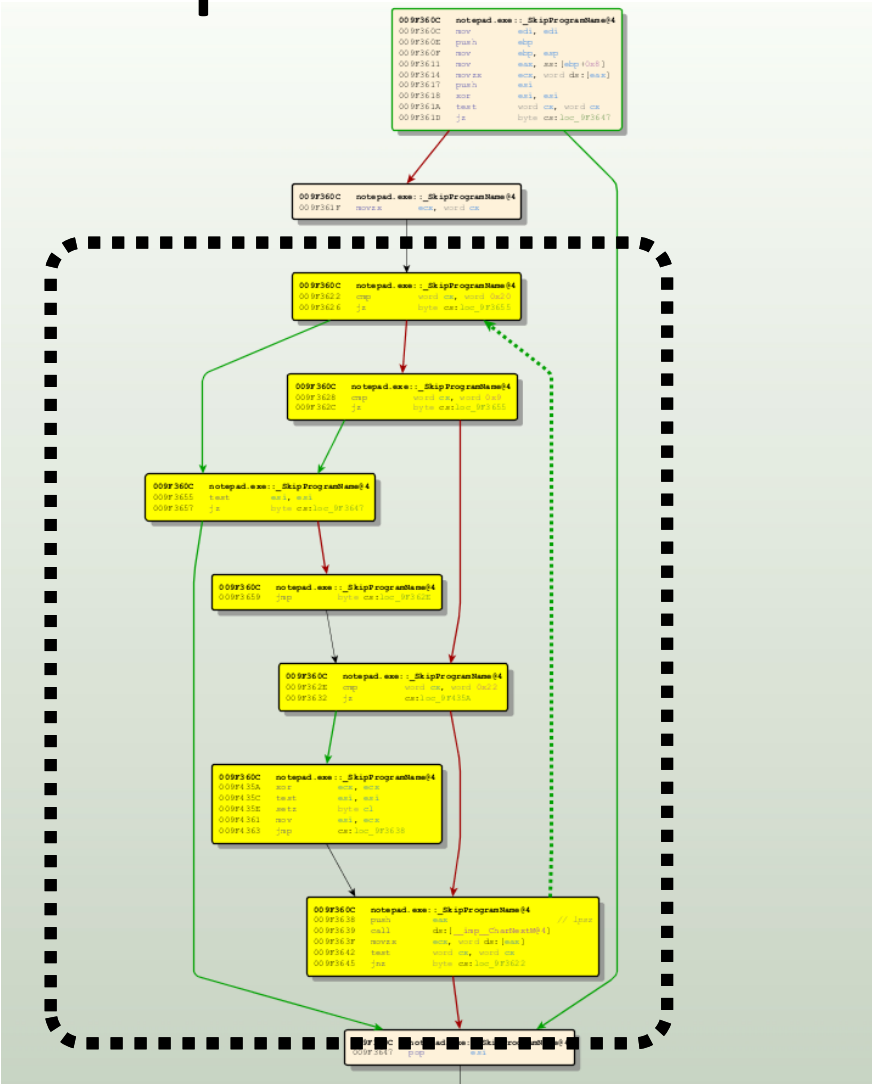
*This graph is **not** strongly connected.*

SCC



*But it contains a **subgraph** which is strongly-connected.*

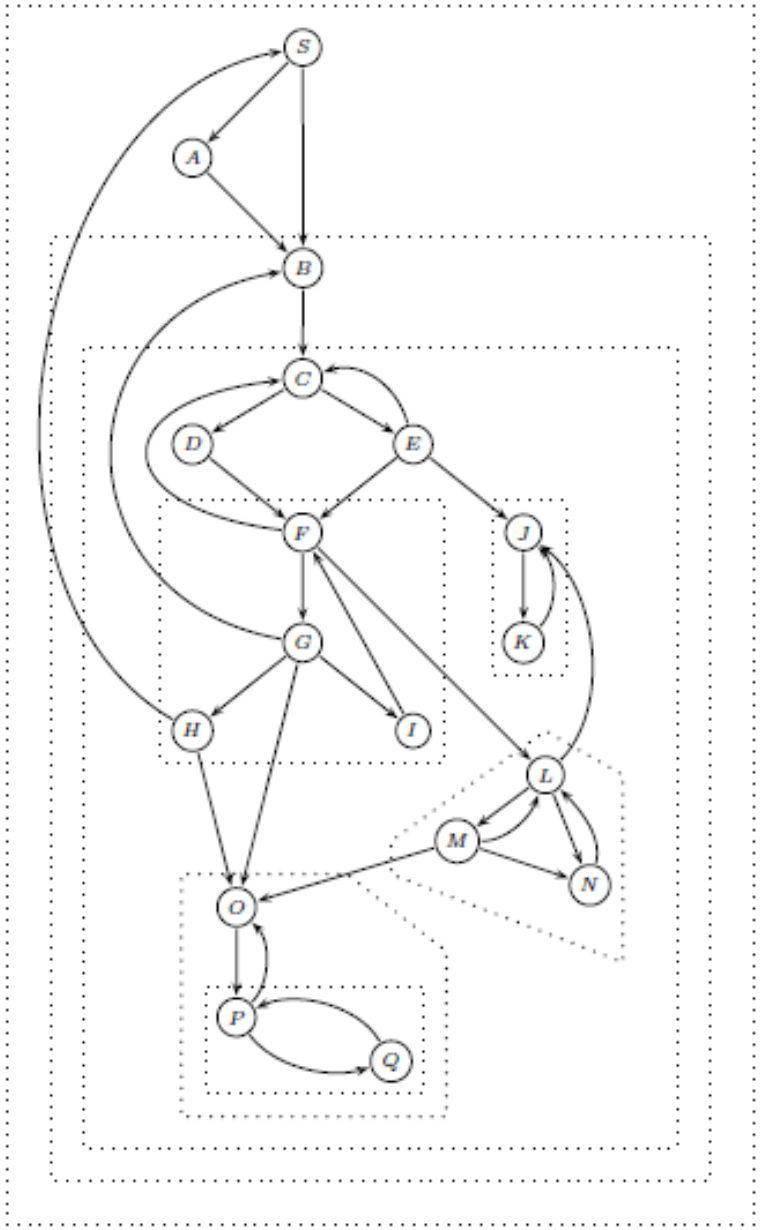
Demo – implemented on BinNavi



Interval analysis

- Unfortunately SCC isn't able to identify ***nested* loops !**
- Interval Analysis
  - Divides the CFG into **regions** and **pre-intervals.**
- To avoid unnecessary details, regions are similar to a **SCC with a unique entry node.**

Nested Intervals



Interval analysis

- DEMO
  - translateString (contains nested loops)
  - mergeStrings
  - compare SCC with Interval analysis

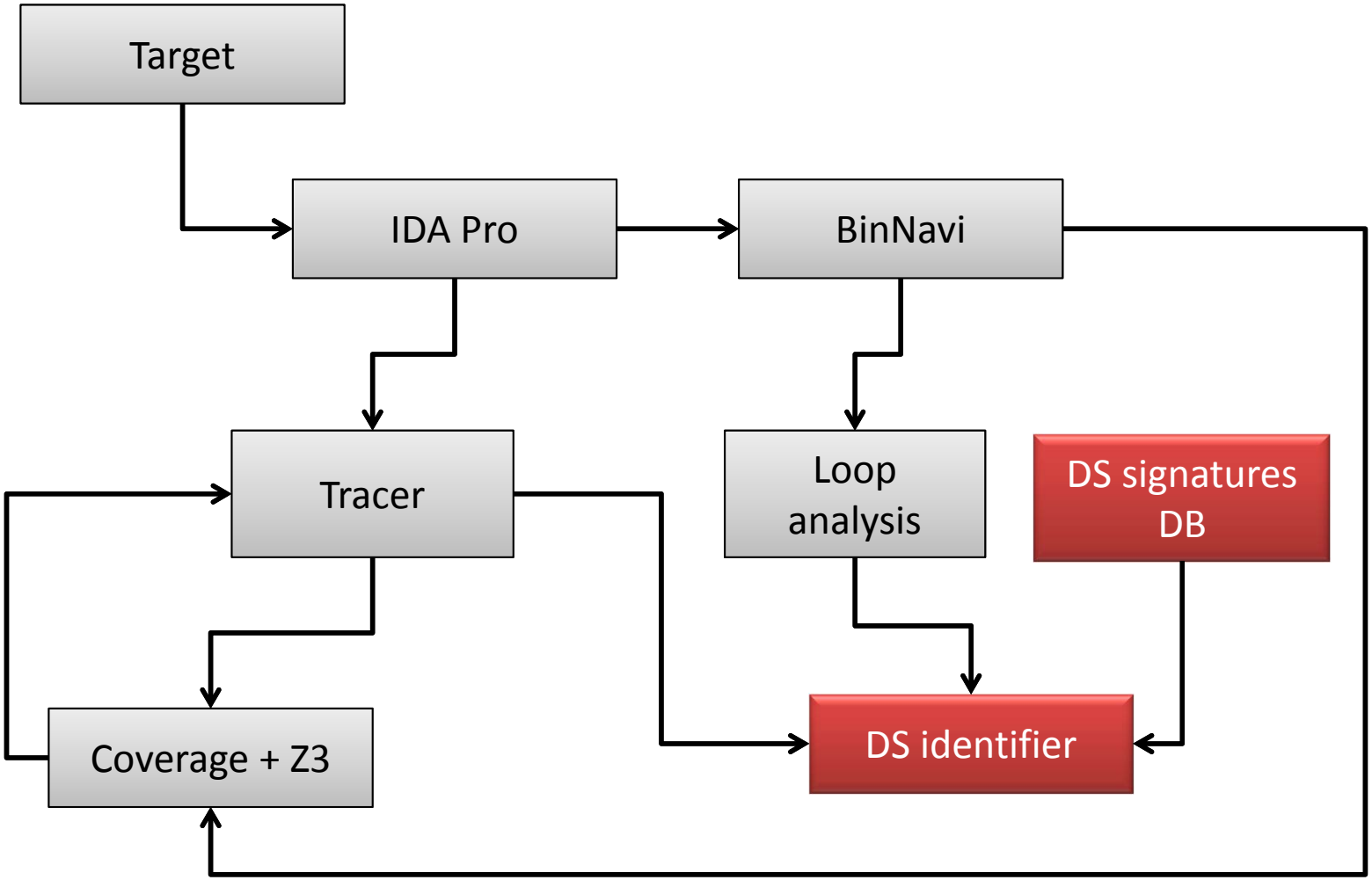
Loop analysis – n-grams

- To extract n-grams and skip-grams of **loops** it is necessary to **walk** the graph and generate the traces.
- For most cases it is possible to generate **all** possible paths of the loop (limited to 2 iterations). This is enough to extract the n-grams (skip-grams)

SyScan'12

DATA STRUCTURE PATTERNS

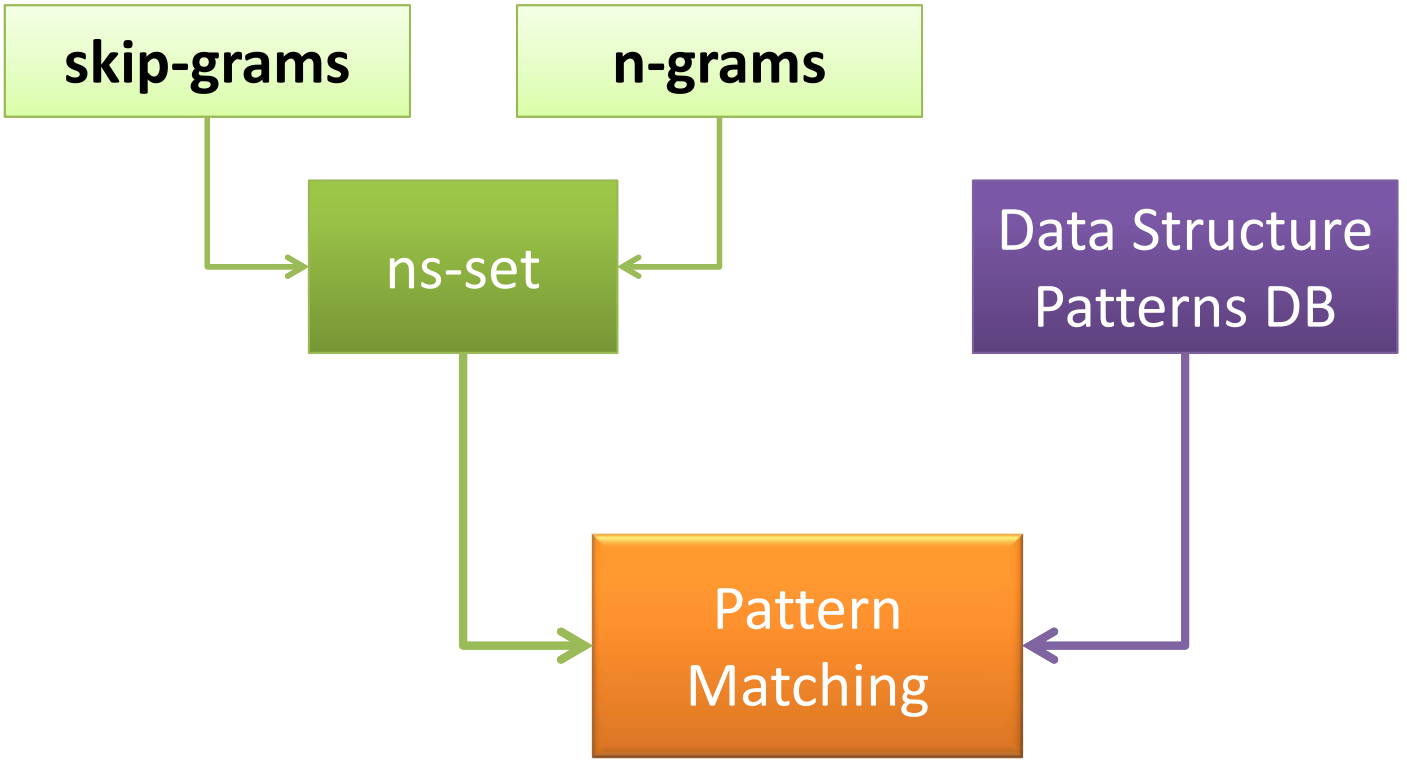
Architecture



DS patterns

- We have:
  - A collection of traces (execution tracer)
  - Coverage increased by Constraint Satisfaction
  - Loop analysis
- We can extract:
  - big sets of n-grams and skip-grams **(ns-set)** from the traces and the loops (filtered by data dependency)

Data Structure identification



Data Structure - patterns

- What defines a linked-list?
- Or a queue, array, stack, tree, graphs and other data structures?

Patterns

- Memory:
  - Memory access order – data dependency
  - Size of the memory operation
  - Base, index, scale and offset from the memory expression
- Operations executed on data:
  - comparisons
    - against zero before another data access? (null-pointer)
    - against a fixed value (signatures/fixed-size)

Pattern matching

- Pattern matching engine to identify the data structures from the n-gram/skip-gram sets.
- Regex style
  - Quantification and Grouping on n-grams
- Accepts queries based on:
  - x86 language
  - REIL language
  - Instruction categories
    - arithmetic
    - branch
    - test
    - data movement (←)

Linked-lists

- Pattern (inside the loop):
  - base on data movement (←)

$$reg_a \leftarrow [reg_b + offset]$$

…

(skipped bytes)

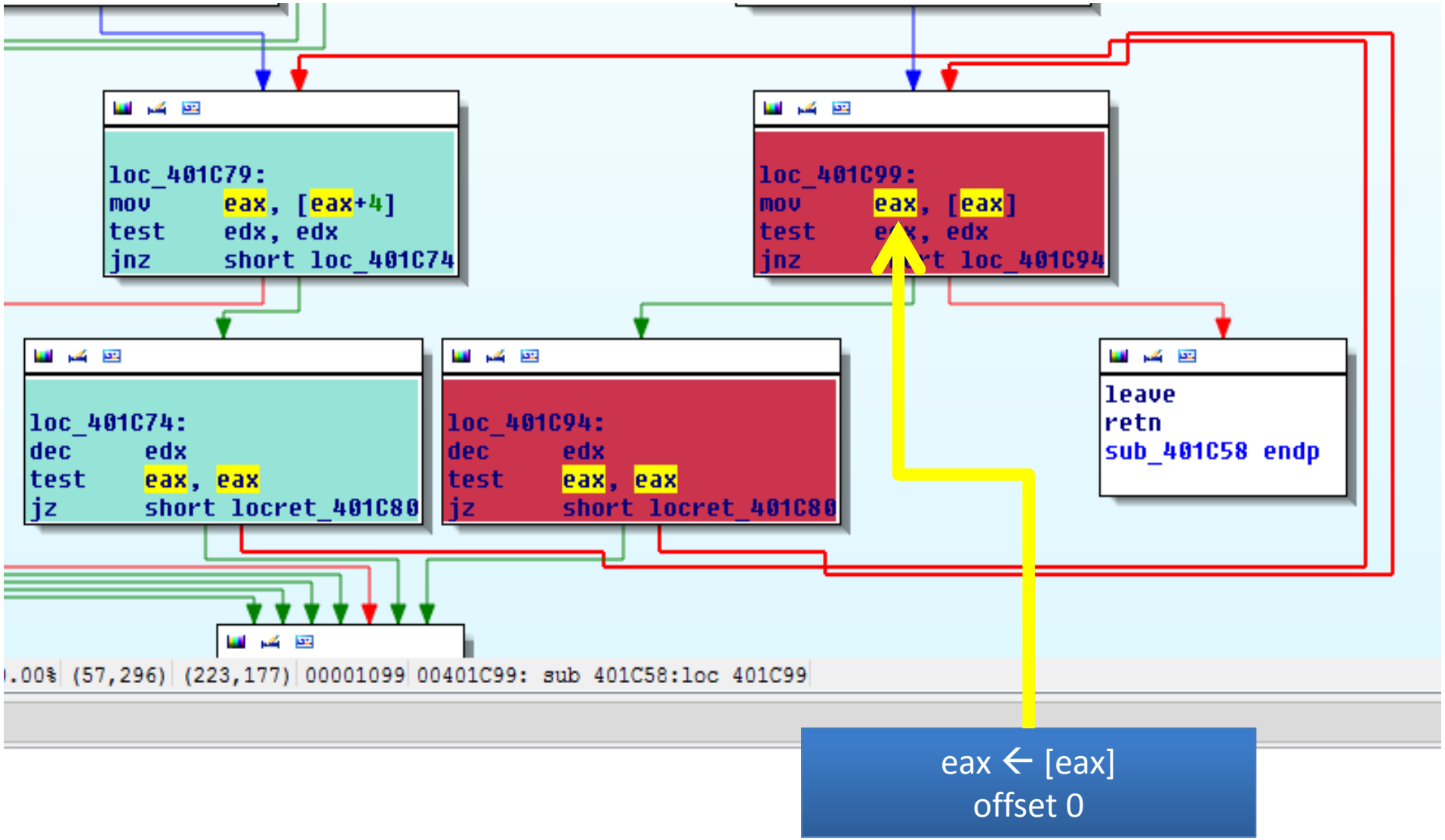
…

$$reg_b \leftarrow reg_a$$

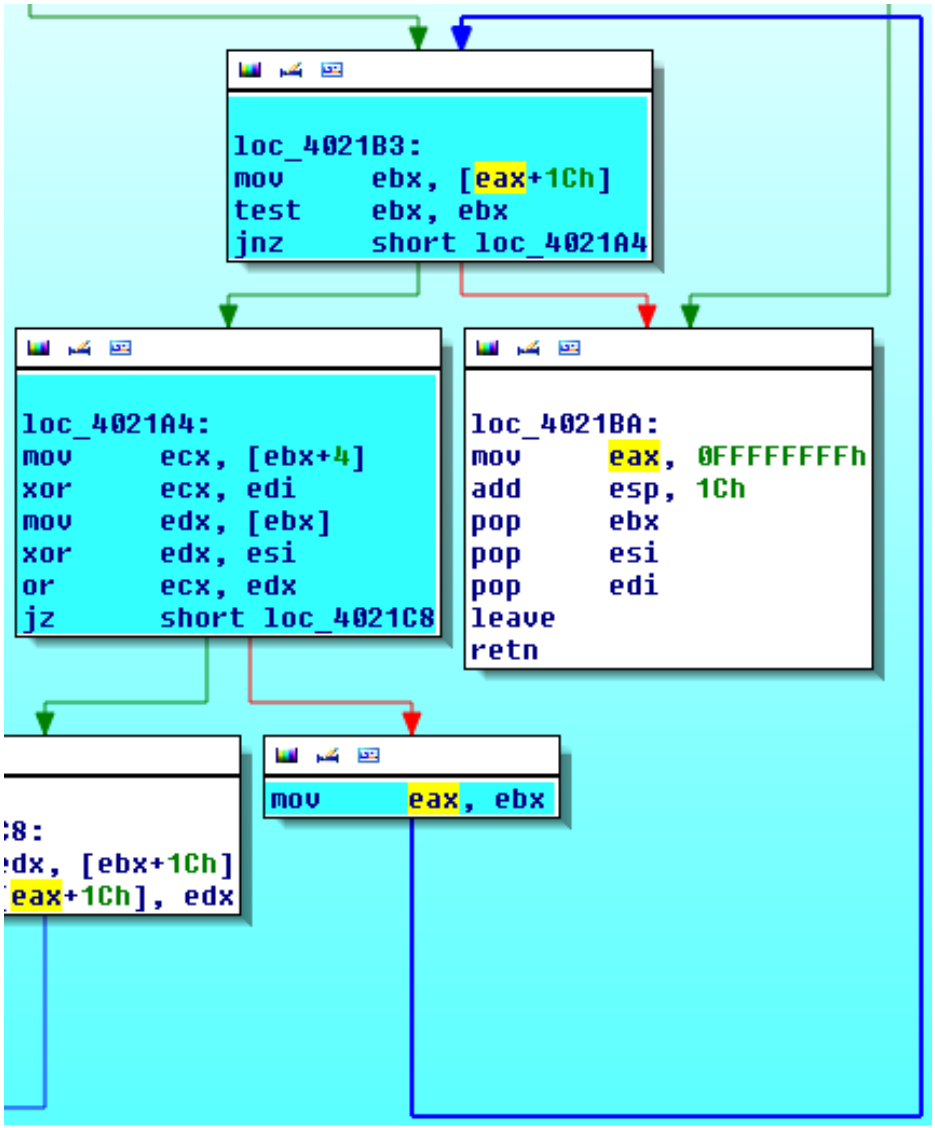
Linked-lists

- Offset
  - Indicates the offset of the linked-list pointer field inside the structure.
- Cycle structure
  - The loop generates a cycle structure connecting the instructions

redis-server linked lists



redis-server



Pattern:

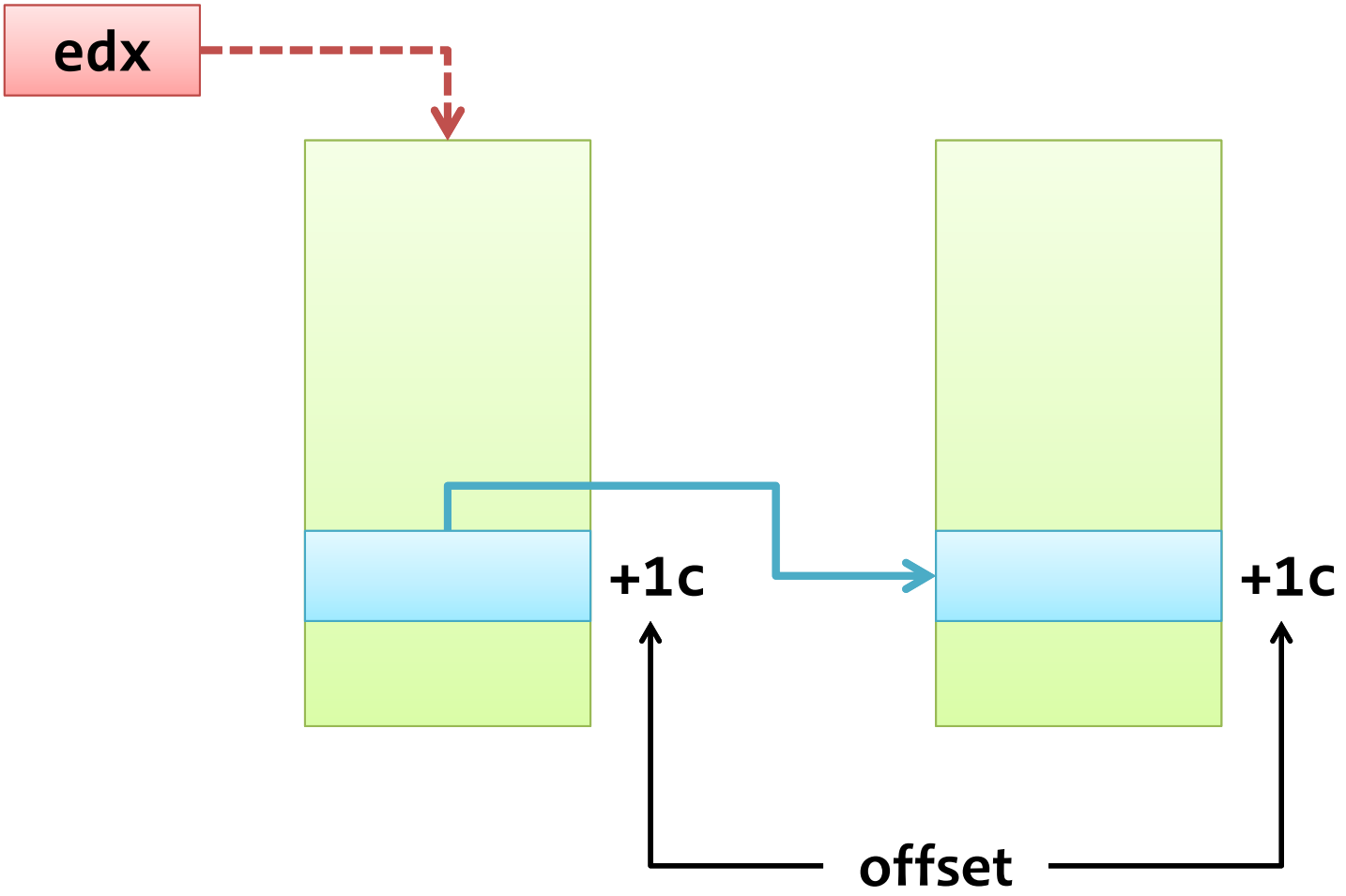
[eax+1ch] → ebx

... (skipped)

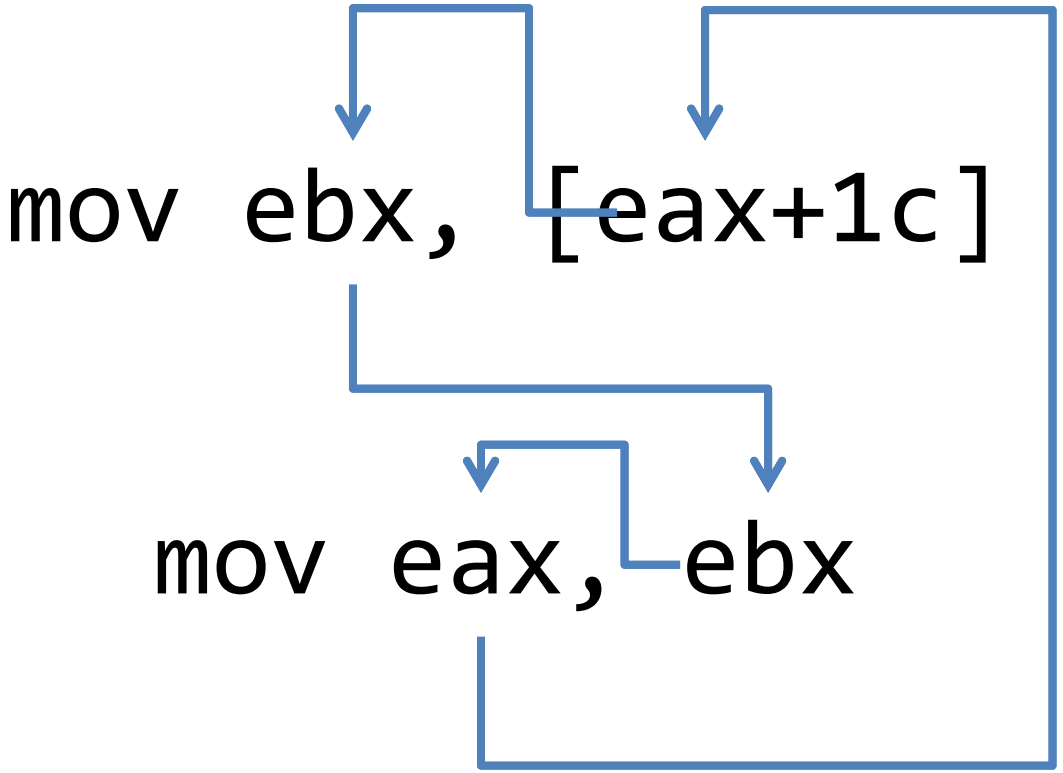
ebx → eax

offset **`0x1c`** of the structure contains a linked list

linked list



Cycle structure



Arrays

- ***Pattern (loop)***

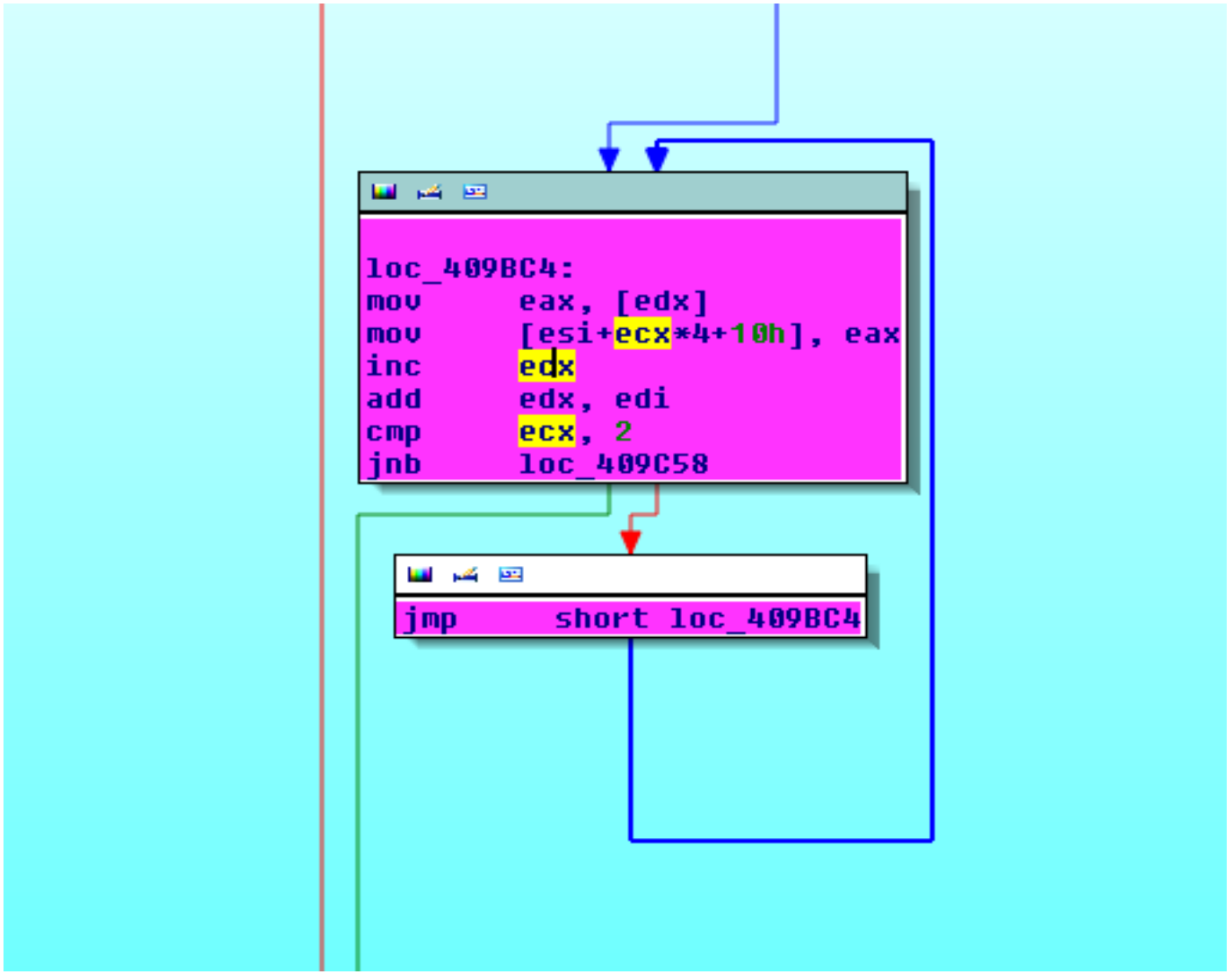
$$reg_r \leftarrow [reg_b + reg_{index} * scale + offset]$$

…

$$inc\ reg_{index}$$

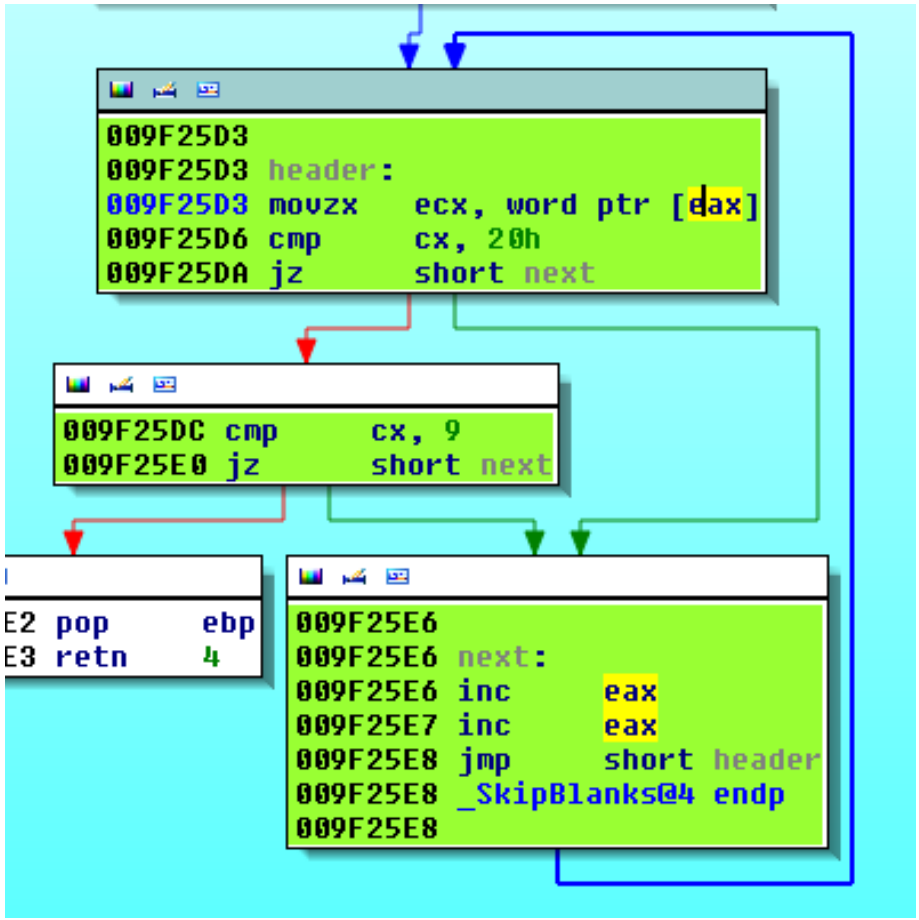
- offset optional
- it can also be a sequence of more than one **inc** or any other arithmetical operation.

Array detection



ntoskrnl!WmipFirmwareTableHandler()

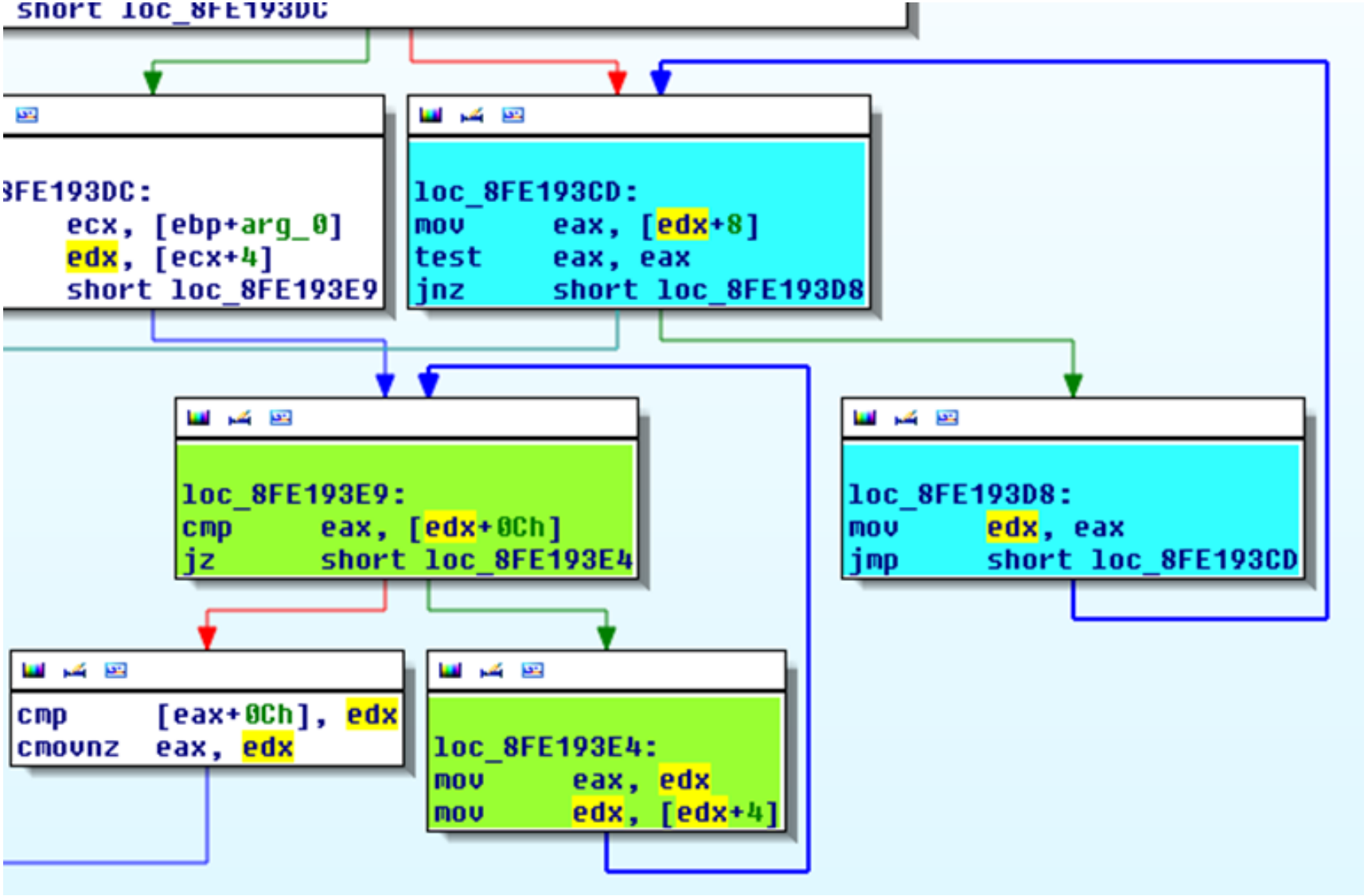
Strings – Unicode string



Data structure signatures

- Tree implementation is based on linked lists!
  - Detect linked lists first
  - Operation: Comparison of values
    - Determines the offset of the next structure field access

Mac OS - dyld


short loc_8FE193DC
3FE193DC:
ecx, [ebp+arg_0]
edx, [ecx+4]
short loc_8FE193E9
loc_8FE193CD:
mov eax, [edx+8]
test eax, eax
jnz short loc_8FE193D8
loc_8FE193E9:
cmp eax, [edx+0Ch]
jz short loc_8FE193E4
loc_8FE193D8:
mov edx, eax
jmp short loc_8FE193CD
cmp [eax+0Ch], edx
cmovnz eax, edx
loc_8FE193E4:
mov eax, edx
mov edx, [edx+4]

SyScan'12

FINAL SECTION

Accuracy

- The detection is probabilistic
- Something detected as a Linked List can be a Tree data structure.
- Increasing accuracy factors:
  - **Code coverage**
  - **Quality** of the data structure pattern descriptions for the matching engine.

Future

- **Work in progress!**
- Publish the data structure patterns DB!
- Finish the **pattern matching engine**
- DSL for DS pattern description.
- Web interface/API for queries
- Type propagation for detected data structures
- Create a **type information** database for **syscalls.**

Future

- Taxonomy for data structures
- Publish list of most common n-grams (statistics)
- Create a taxonomy for data structures

References

- Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python* (1st ed.). O'Reilly Media, Inc.
- Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.
- http://books.google.com/ngrams
- All Our N-gram are Belong to You http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html
- David Guthrie and Ben Allison and Wei Liu and Louise Guthrie and Yorick Wilks, A Closer Look at Skip-gram Modelling

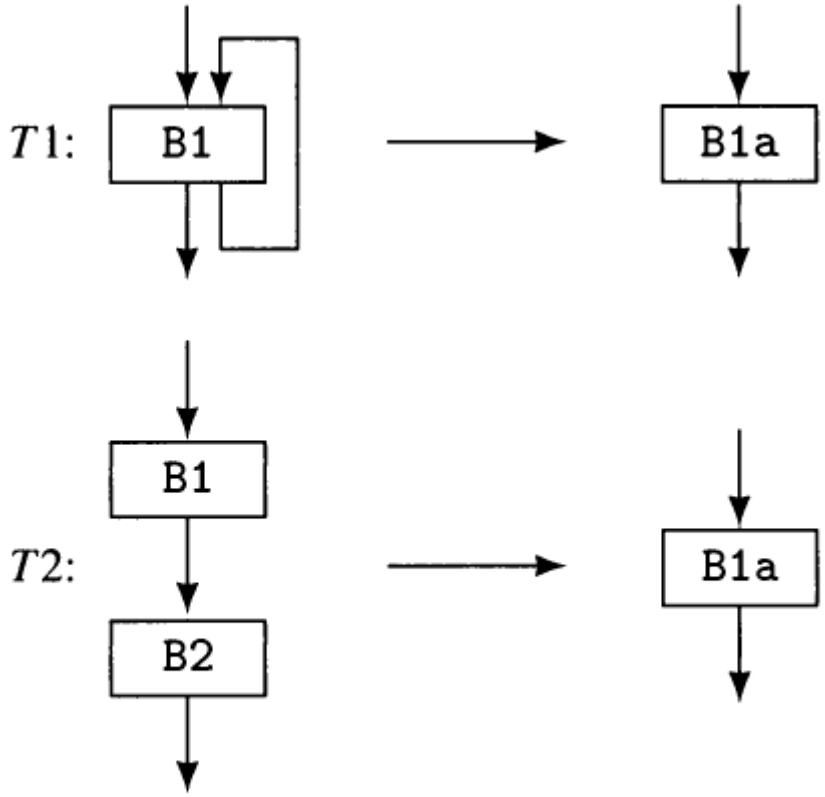
QUESTIONS?

SyScan'12

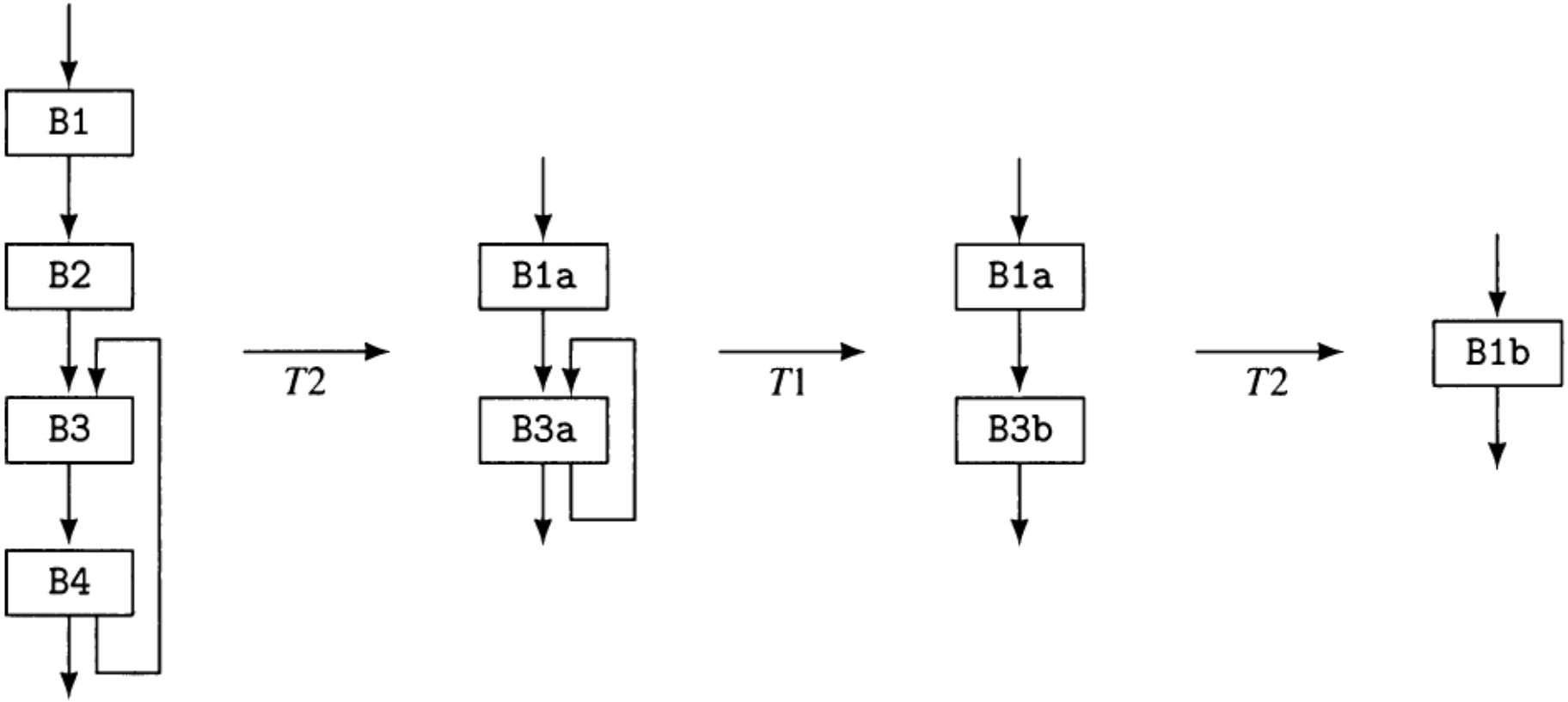
EXTRA SLIDES

T1/T2 transformations

- T1/T2 → 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.

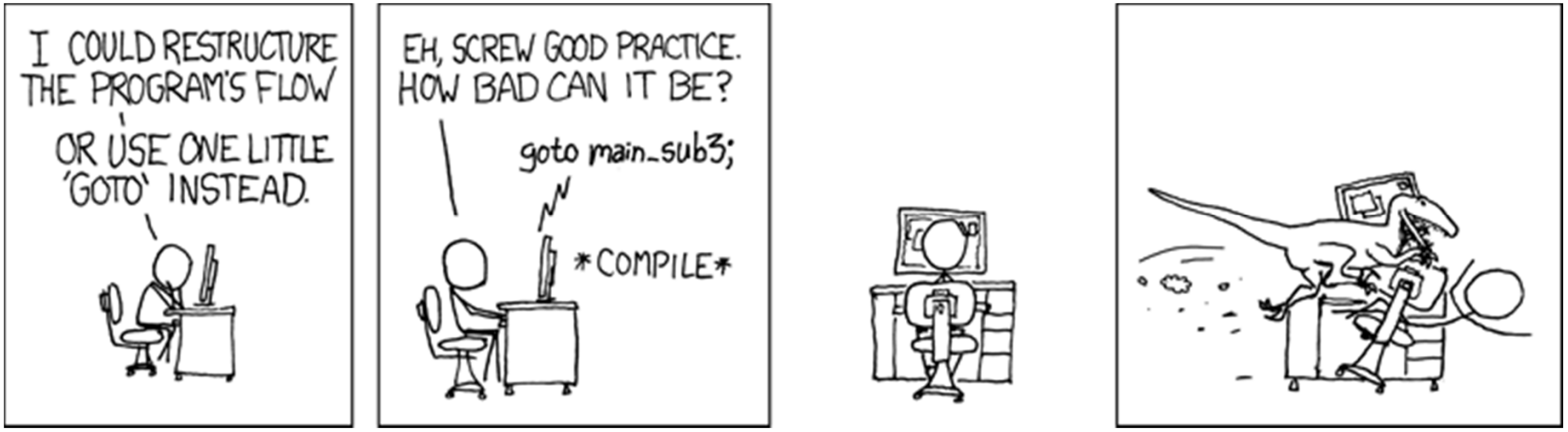


$T1$-$T2$ transformations.



Example of $T1$-$T2$ transformations.

GOTO considered harmful…

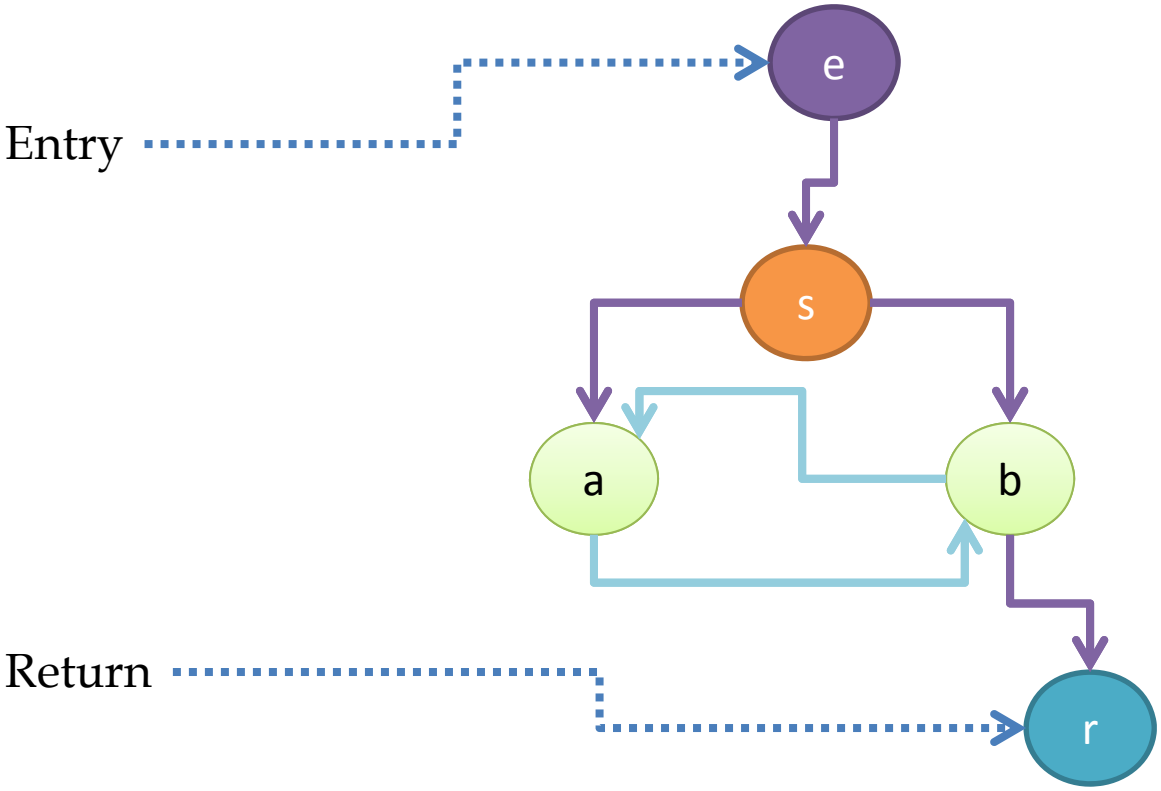


http://xkcd.com/292/

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

- What kind of code produces irreducible graphs?
  - 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 to the body of a WHILE statement

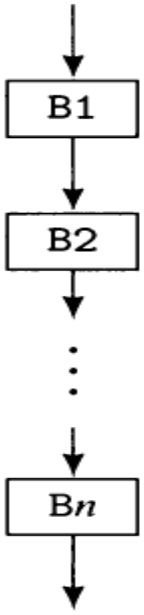
Solutions

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

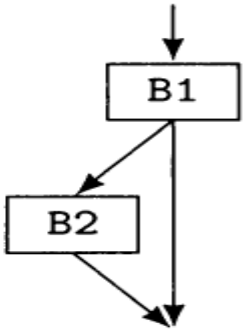
Structural Analysis

- 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)

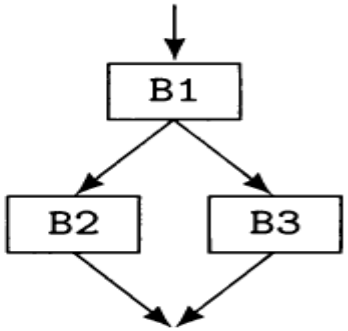
Acyclic schemas



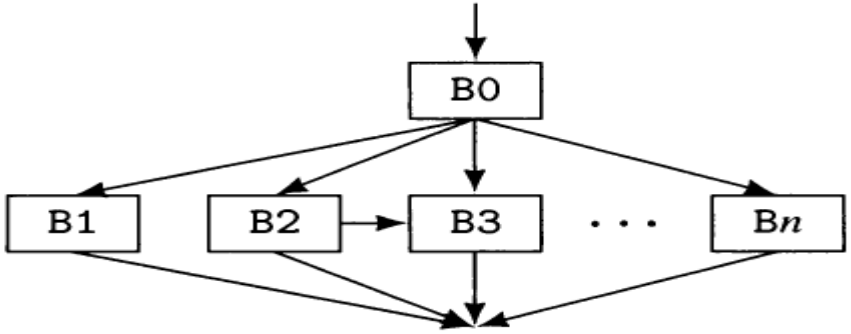
block schema



if-then

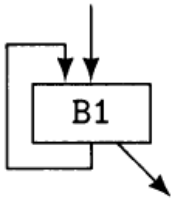


if-then-else

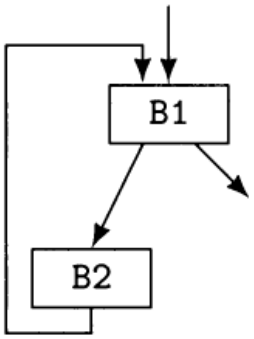


case/switch schema

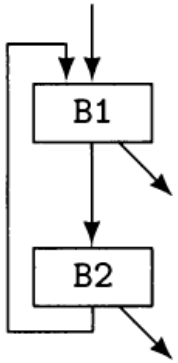
Cyclic schemas



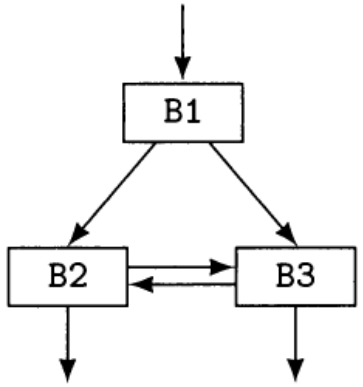
self loop



while loop



natural loop schema

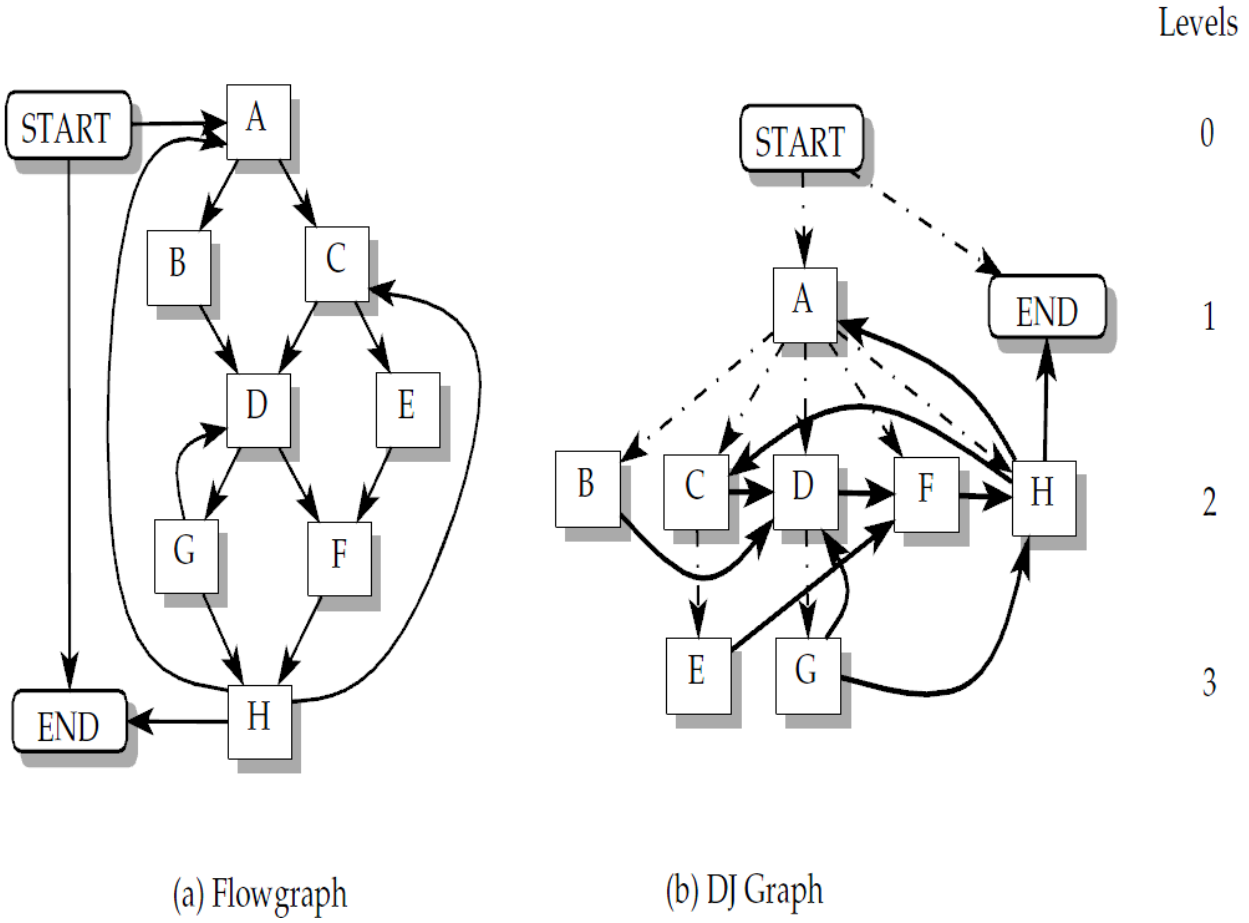


improper interval schema

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



(a) Flowgraph

(b) DJ Graph