“More fun with Graphs”

Halvar Flake – Blackhat Federal 2003

Outline for the talk

Structural Function Signatures

- Motivation
- Problems with the signature-based approach
- Heuristics for matching graphs
- Applications

Loop detection in Binaries:

- Detecting simple loops
- Determining if a loop copies memory
- Estimating the number of iterations a loop can take

Function Signatures (I)

What are function signatures ?

- Functions in disassemblies originally have no names, just addresses
- Function signatures allow automatically retrieving names for known functions
- Function signatures are mainly used to recognize statically linked libc functions
- Massive aid in disassembling – who would want to spend his time finding _malloc() or strcpy() manually ?

Function Signatures (II)

What else are function signatures good for ?

- Porting information in disassemblies to a new version (e.g. porting info from an existing Disassembly of FW-1 to an updated version)
- Scanning binaries for known-to-be vulnerable libs (zlib ☺)
- Finding functions under GPL in closed-source, commercial applications
- Porting debug info which vendors accidentally left in an old executable to new versions of the program
- Finding differences between two different releases of the same file (Microsoft Security Patches ☺)

Function Signatures (III)

Usual approach to signatures:

Pattern matching with wildcards

- IDA's FLIRT system
  - IDB_2_PAT
  - IDB_2_SIG
- Fenris signature system

Problems

- Normal pattern matching is problematic

  → A few lines of code that change can lead to different register allocation and thus to many changed locaitons

  → A few lines of code that change can lead to basic blocks having different sizes and ending up in completely different places (MS internal optimization)

- A small change can produce two binaries which hardly resemble each other

Solution ?

- Structural fingerprinting ?

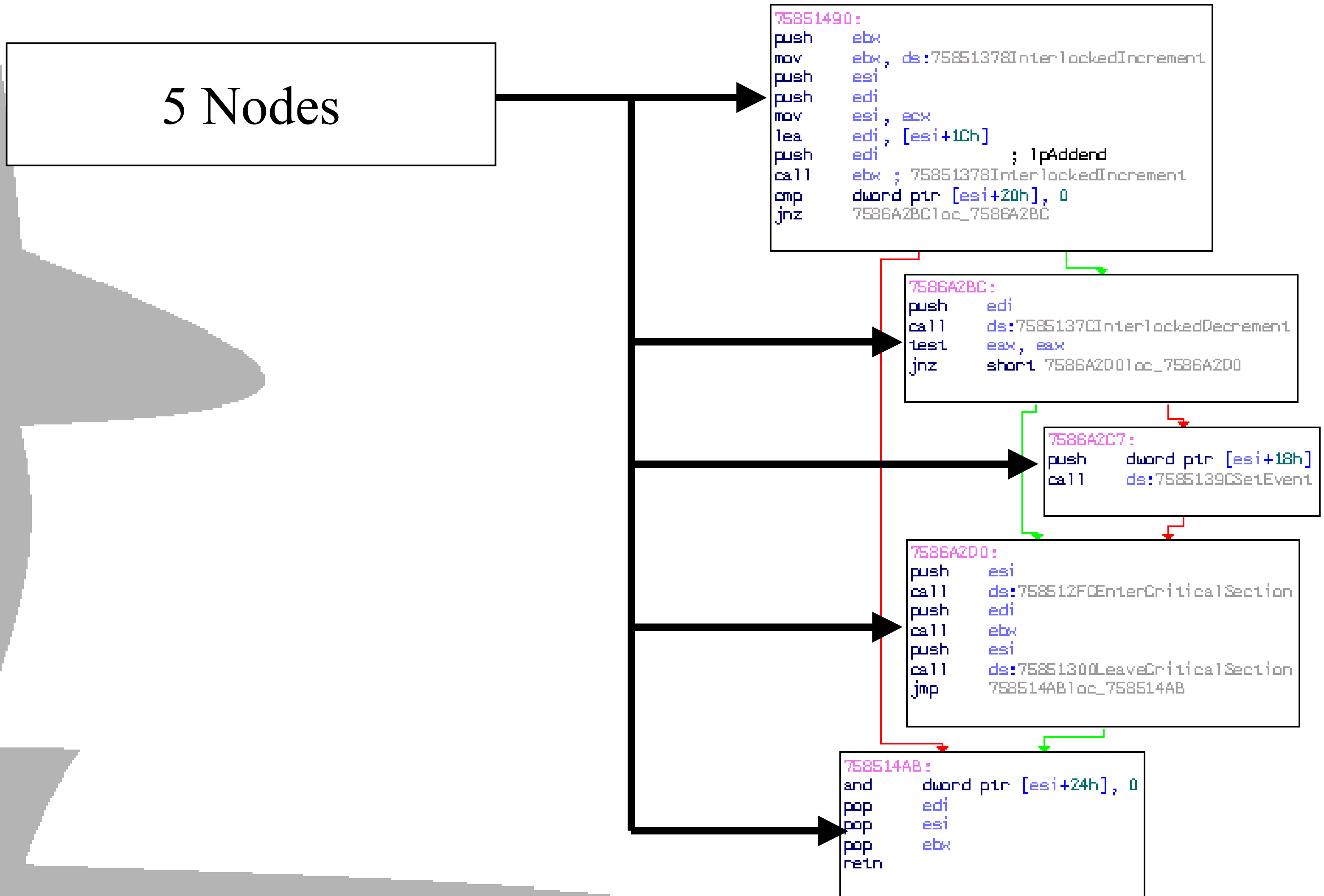
  → Function flowgraphs will stay the same, regardless of register allocation or basic block reordering

- Graph Isomorphisms (math-speak for finding out if two graphs are the same) are computationally expensive to compute

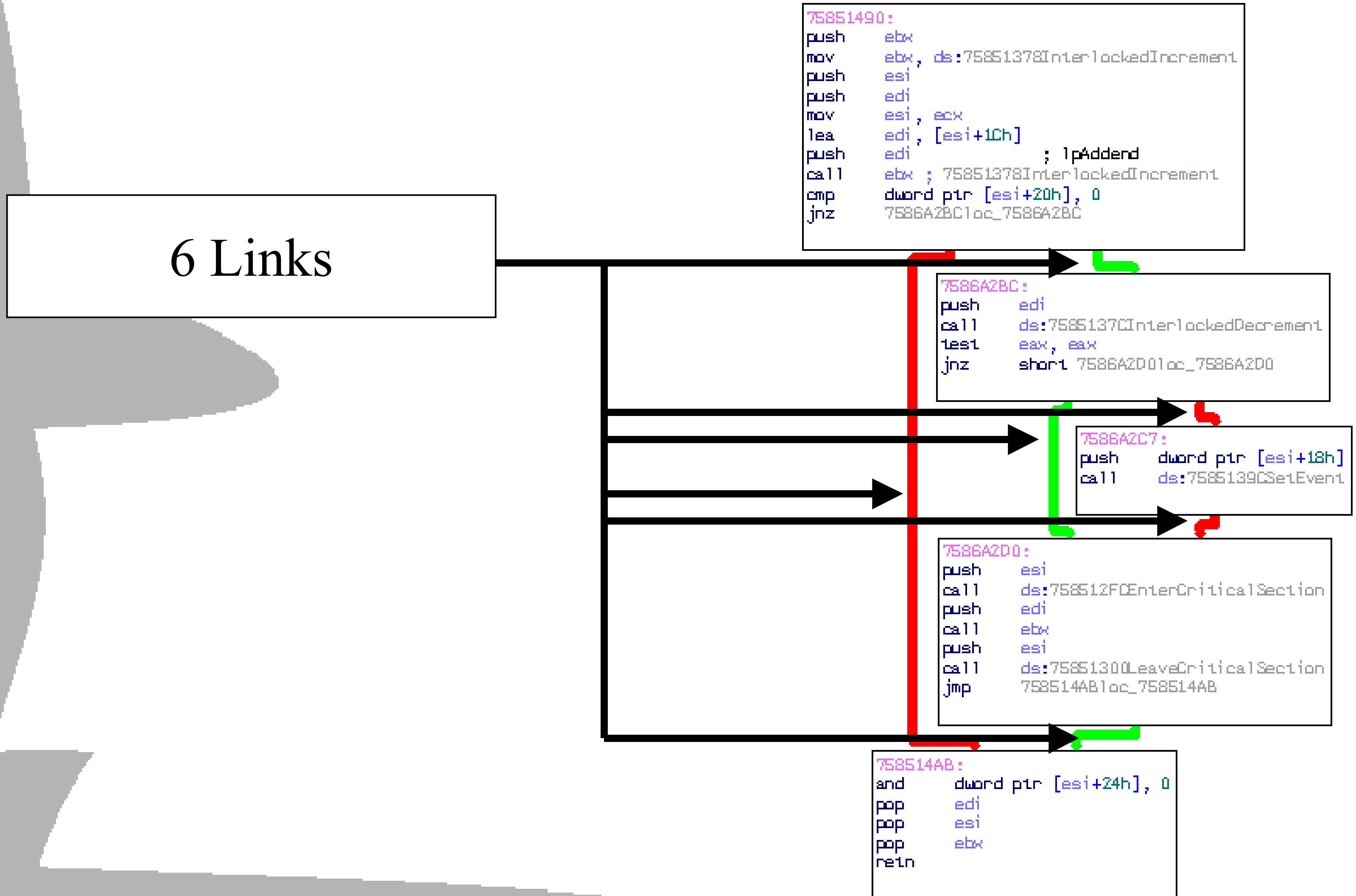
→ A simpler solution (using matching heuristics) can yield usable results

→ Comparing number of code blocks, number of links and number of subfunction calls

Example (I)

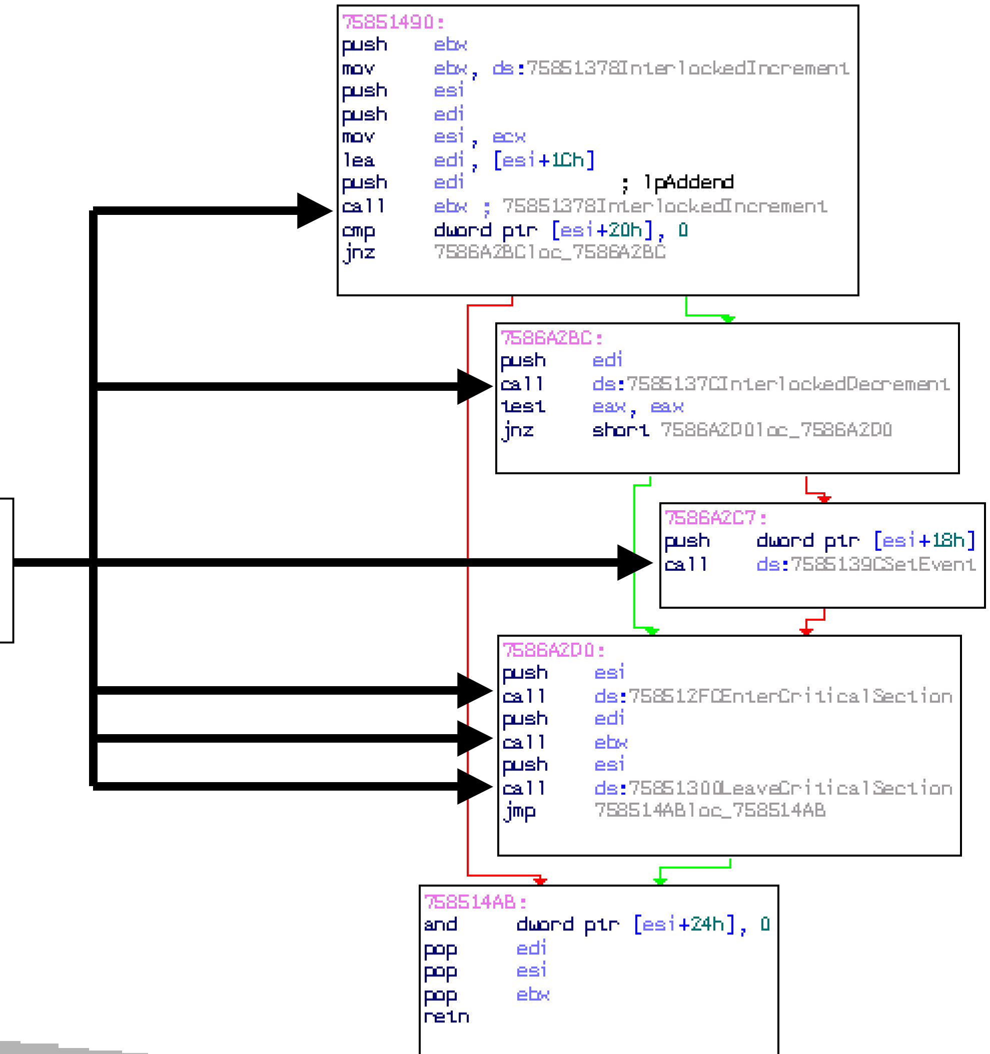

5 Nodes
75851490:
push ebx
mov ebx, ds:75851378InterlockedIncrement
push esi
push edi
mov esi, ecx
lea edi, [esi+1Ch]
push edi ; lpAddend
call ebx ; 75851378InterlockedIncrement
cmp dword ptr [esi+20h], 0
jnz 7586A2BCloc_7586A2BC
7586A2BC:
push edi
call ds:7585137CInterlockedDecrement
test eax, eax
jnz short 7586A2D0loc_7586A2D0
7586A2C7:
push dword ptr [esi+18h]
call ds:7585139CSetEvent
7586A2D0:
push esi
call ds:758512FCEnterCriticalSection
push edi
call ebx
push esi
call ds:75851300LeaveCriticalSection
jmp 758514ABloc_758514AB
758514AB:
and dword ptr [esi+24h], 0
pop edi
pop esi
pop ebx
retn

Example (II)


6 Links
75851490:
push ebx
mov ebx, ds:75851378InterlockedIncrement
push esi
push edi
mov esi, ecx
lea edi, [esi+1Ch]
push edi ; lpAddend
call ebx ; 75851378InterlockedIncrement
cmp dword ptr [esi+20h], 0
jnz 7586A2BCloc_7586A2BC
7586A2BC:
push edi
call ds:7585137CInterlockedDecrement
test eax, eax
jnz short 7586A2D0loc_7586A2D0
7586A2C7:
push dword ptr [esi+18h]
call ds:7585139CSetEvent
7586A2D0:
push esi
call ds:758512FCEnterCriticalSection
push edi
call ebx
push esi
call ds:75851300LeaveCriticalSection
jmp 758514ABloc_758514AB
758514AB:
and dword ptr [esi+24h], 0
pop edi
pop esi
pop ebx
retn

Example (III)

6 subcalls



Signature: 5/6/6

Pro / Con

Advantages:

- Tolerant to basic block reordering
- Tolerant to differences in register assignments
- Will find all structural changes (e.g. an added if( ))
- Reasonably "sharp" for larger functions

Disadvantages:

- Will not find changes in constant values
- Will not find changes in buffer sizes
- No useful signature for very small functions can be generated (1/0/0 will be the signature for every very simple functino)

Open Source Patches

Open Source Patches:

- Visible to everyone → Publicising the patched version makes the bug (or bugclass) public
- Many people regularly read CVS updates like others read the newspaper → Security-critical changes cannot "sneak in"
- Many eyes make bugfixes thorough → Changes that fix the "symptom" but not the root cause are rare

→ Keeping bug information quiet after publication of a patch is next to impossible

Closed Source Patches

Closed Source Patches:

- Vendors try to keep details of bugs silent
  "No need to tell the hackers what is going on"
- Vendors underestimate impact of bugs:
  "Malformed input leads to disclosure of hexadecimal values from memory"
  [Euphemism for format string bug]
  "This problem can lead to a DoS-style-attack"
  [Euphemism for remotely exploitable bug in Ring-0 code]
- Vendors try to "piggyback" security patches onto less interesting patches

Binary Diff Methodology

We can use these signatures to detect which changes a vendor undertook with a security patch:

- Generate all signatures for all functions in both files
- Find “Fixed Points”, e.g. functions whose signature exists only once in both files (roughly _ of all funcs)
- Use the “fixed points” and a calltree to assign the other _ of all signatures
- Functions that cannot be mapped are candidates that might have changed

(Demonstration)

Questions ?

Loop detection

- Some vendors (MS) have started to have their code audited for bugs
- The focus seems to have been on eliminating strcpy() and other known dangerous library calls
- How could the DCOM have slipped by ?

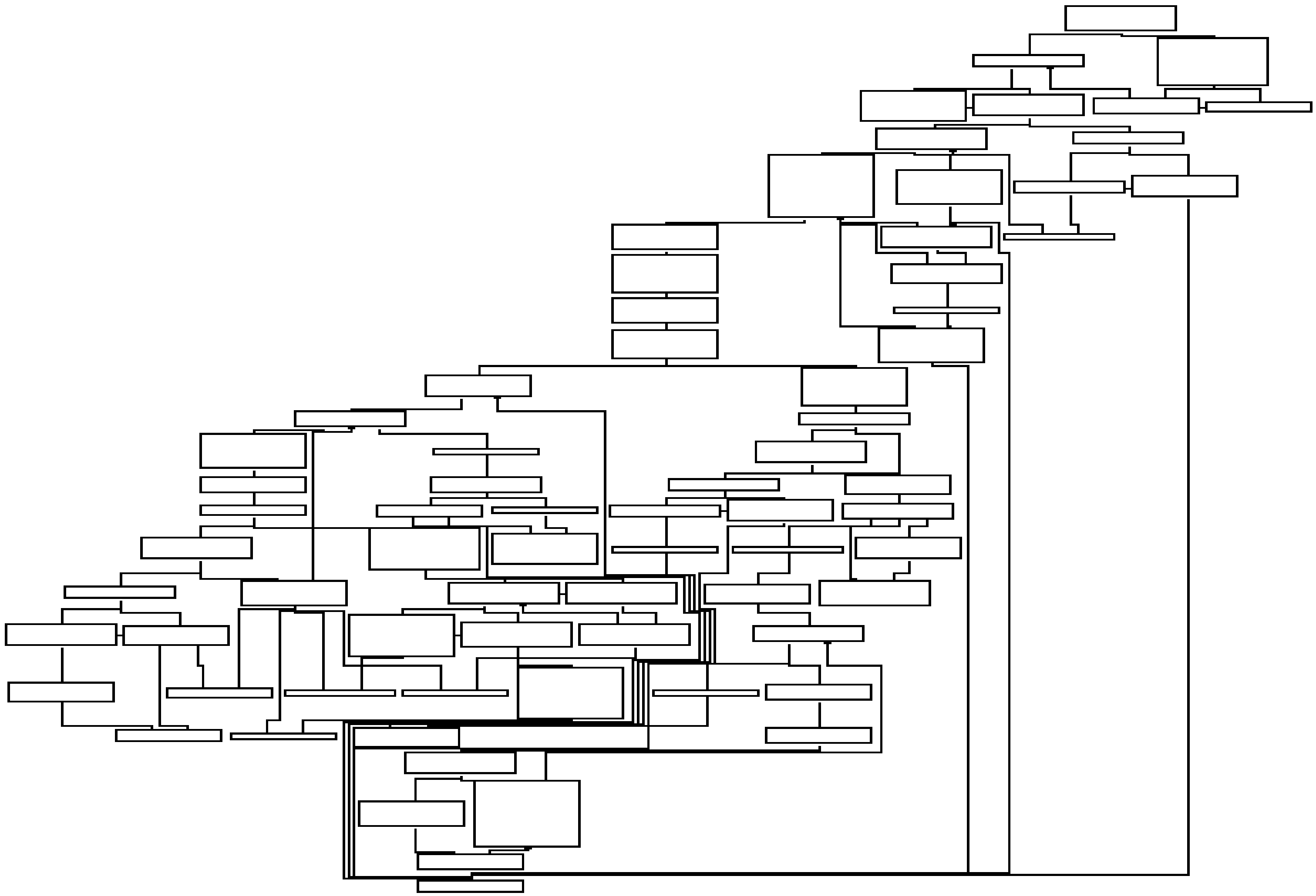
→ Memory – copying loops (decoding etc) seem to have been neglected

- “Loops ? That is so 1998 !” ☺
- Loops are not all that obvious to spot in binaries

→ A mechanism to spot loops in binaries is useful

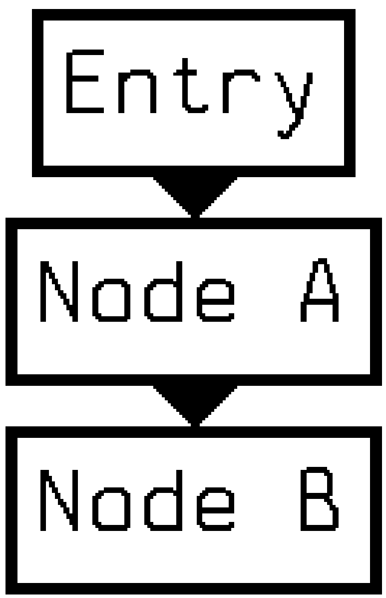
Loop detection (II)

Can you spot the loops ?



Dominator Trees

- A node *A* in a directed graph **dominates** a node *B* if all paths from the entry to node *B* pass through node *A*

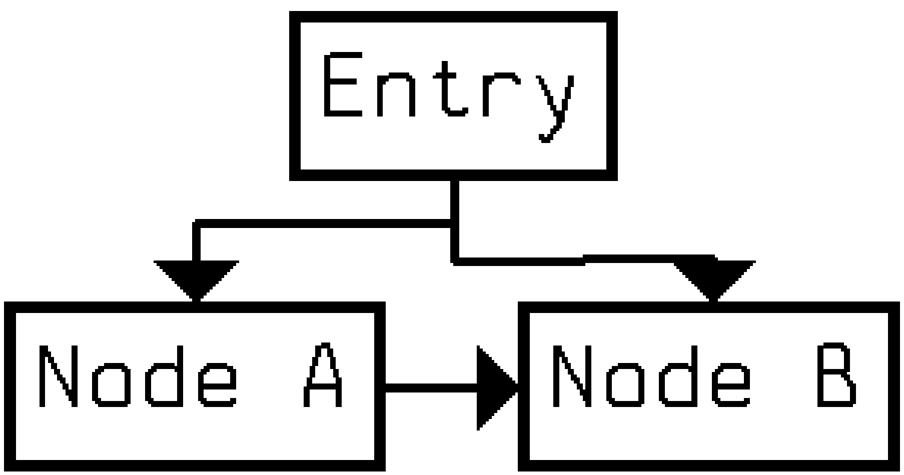


B is dominated by *Entry* and also by *A*

Dominator Trees (II)

- A node *A* in a directed graph **dominates** a node *B* if all paths from the entry to node *B* pass through node *A*

B is dominated by *Entry* but **not** by *A*



Loop detection (III)

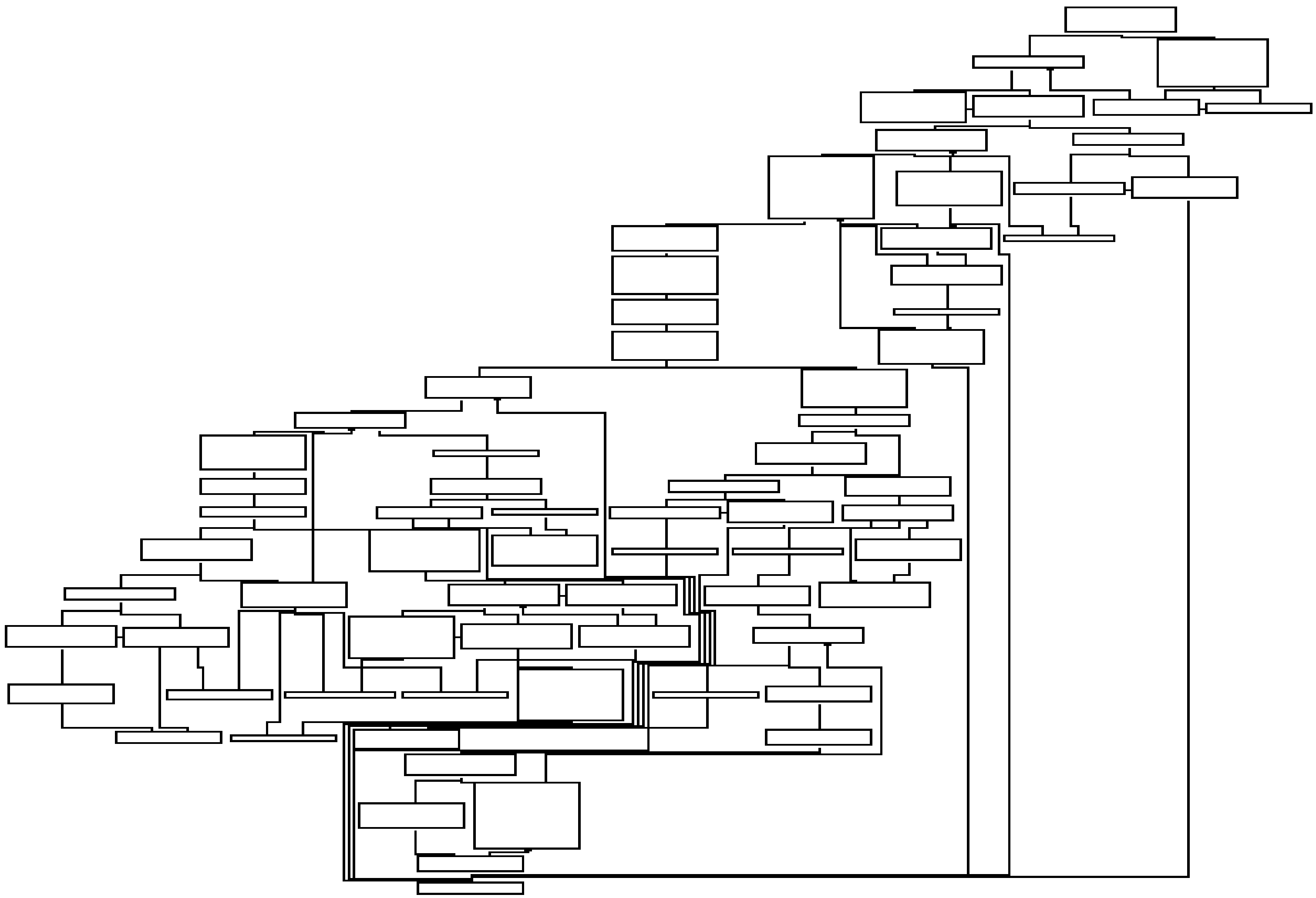
- Dominator Trees can be used to detect loops in graphs
- If a node *A* links to a node *B*, and if *B* dominates *A*, the link closes a loop in the graph

→ All paths to *A* lead through *B*
→ *A* links down to *B*, and all paths to *A* must've run through node *B* ➔ we have found a loop

We can easily build dominator trees from the functions in the binary and thus quickly find loops

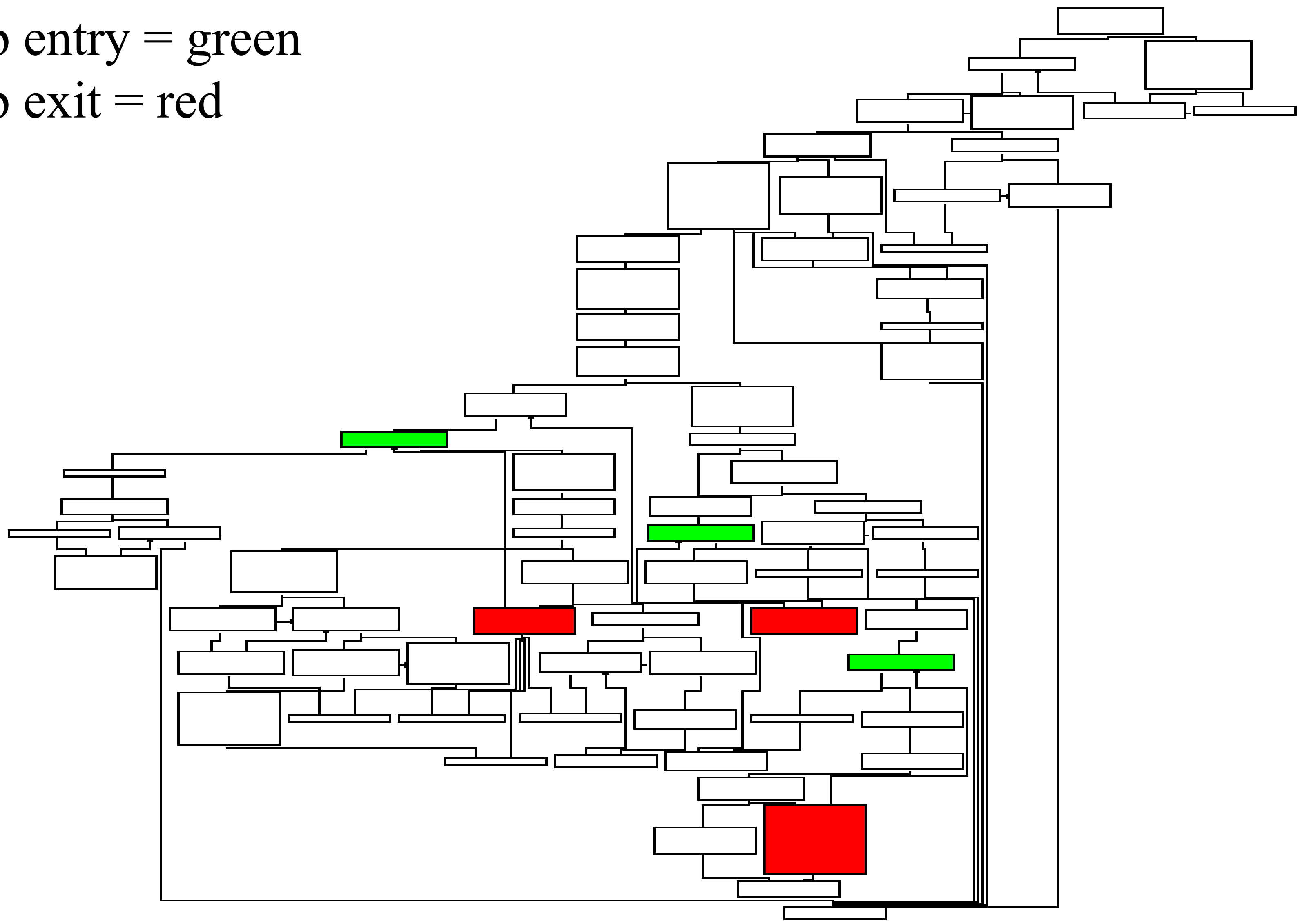
Loop detection (IV)

Can you spot the loops ?



Loop detection (V)

Loop entry = green
Loop exit = red



Killing false positives

- Not all loops are of interest for us
- Loops that do not write to any memory are not interesting
- Loops that just write well-defined variables are not interesting
- Loops that write a statically defined number of bytes are not terribly interesting

→ We want to eliminate all loops that do not write memory

→ We want to eliminate all loops that write to well-defined variables

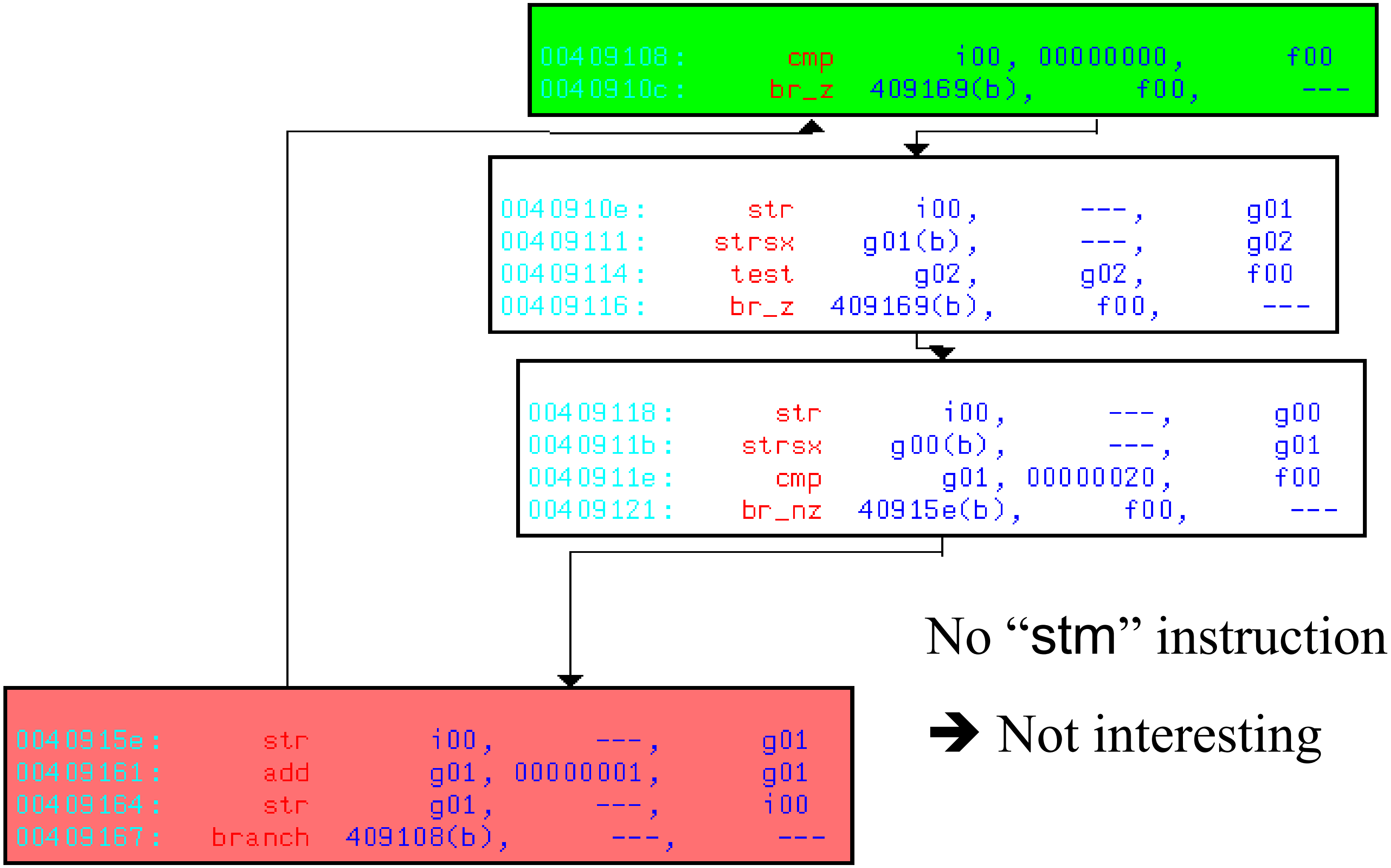
→ We want to eliminate all loops that write a statically defined number of bytes

Memory-Writing

- The examined code has been translated to the MCPU code presented in the last talks
- All memory access is explicit, e.g. there is an explicit instruction for storing memory

→ All loops that do not store stuff into memory can be eliminated by scanning for a "stm" instruction

Memory-Writing (II)



No "stm" instruction

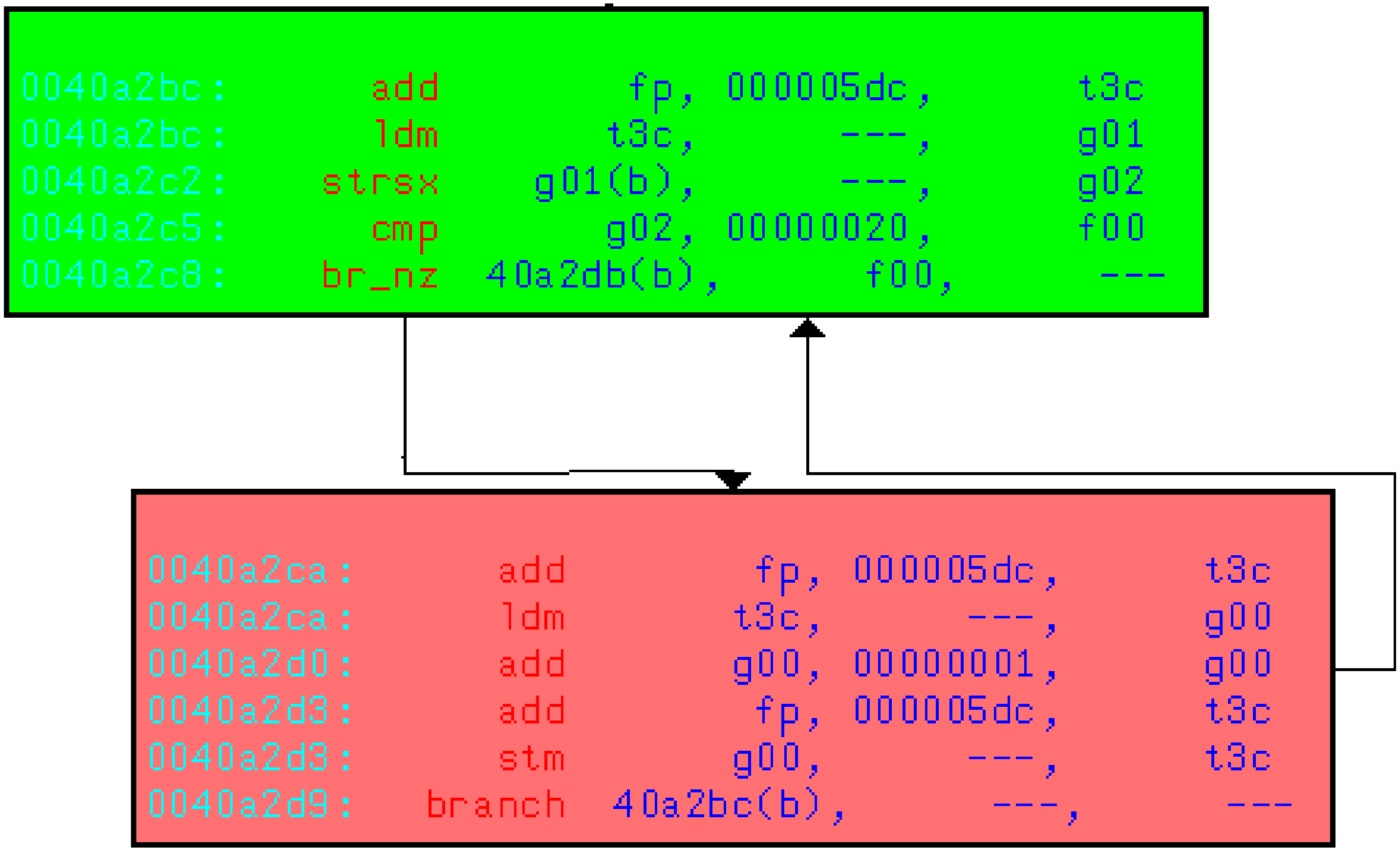
➔ Not interesting

Variable-Writing

- A write access occurs in our loop
- If on every loop iteration, the location it writes to is the same, it is not a memory-copying loop
- If the loop writes to a location like “register + offset” with a hardcoded offset, it accesses a local variable or structure member

→ All loops that do not write to multiple (and changing) locations can be detected by doing data flow analysis on the memory accesses and seeing if they can change in different loop iterations

Variable-Writing (II)



Temp register t3c is written to → t3c is fp + 0x5DC → fp is unchanged → the memory write is not interesting !

Defined Iterations

- A simple memcpy() with a static number of bytes to copy is not likely to be problematic
- If it was, the program would be nonfunctional anyways if it ever reached the relevant location

→ By eliminating all loops that iterate a predefined/static number of times, we can eliminate all loops that copy a static number of bytes

Defined Iterations (II)

```
75859b76:      str        i00,        ---,       g02
75859b7a:      str        g07,        ---,       t01
75859b7b:      str   0000000a,        ---,       t02
75859b7d:      add        g06,   0000001c,       g06
75859b80:      str        t02,        ---,       g01
75859b81:      str        g02,        ---,       g07
```

```
75859b83:      ldm        g07,        ---,       t04
75859b83:      stm        t04,        ---,       g06
75859b83:      add        g06,   00000004,       g06
75859b83:      add        g07,   00000004,       g07
75859b83:      sub        g01,   00000001,       g01
75859b83:    br_nz   75859b83,        f00,       ---
```

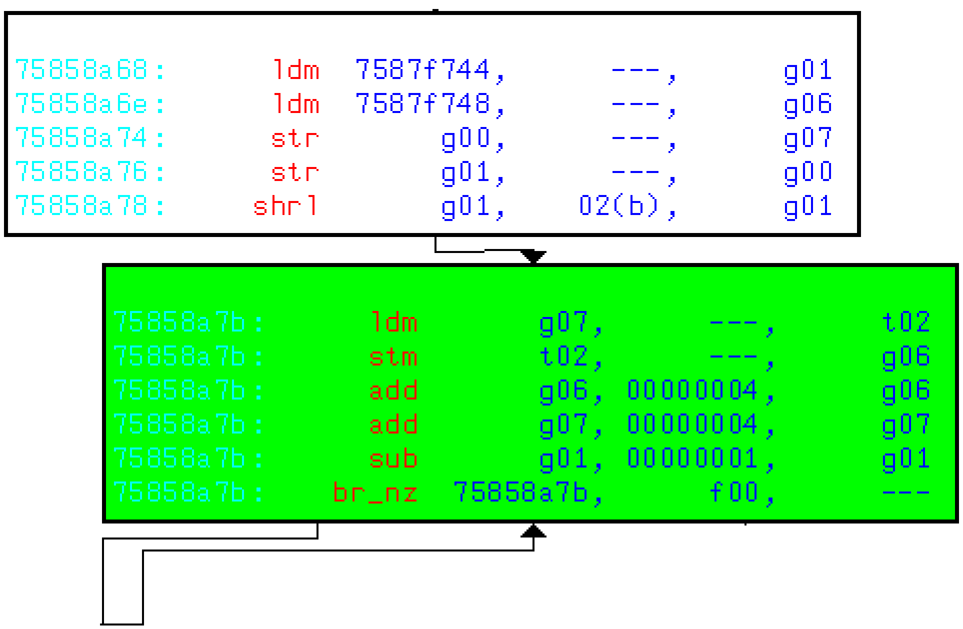
Iterates g01 times

g01 := t02
t02 := 0x0A

➔ Static number of iterations

Summary

- We can automatically detect “interesting” loops, loops that write a dynamically calculated amount of memory
- We can scan multi-megabyte binaries and end up with a dozen or so loops to manually inspect



→ Copies memory

→ iterates an undefined number of times

→ Number of iterations comes from a global variable

➔ Interesting loop

Questions ?