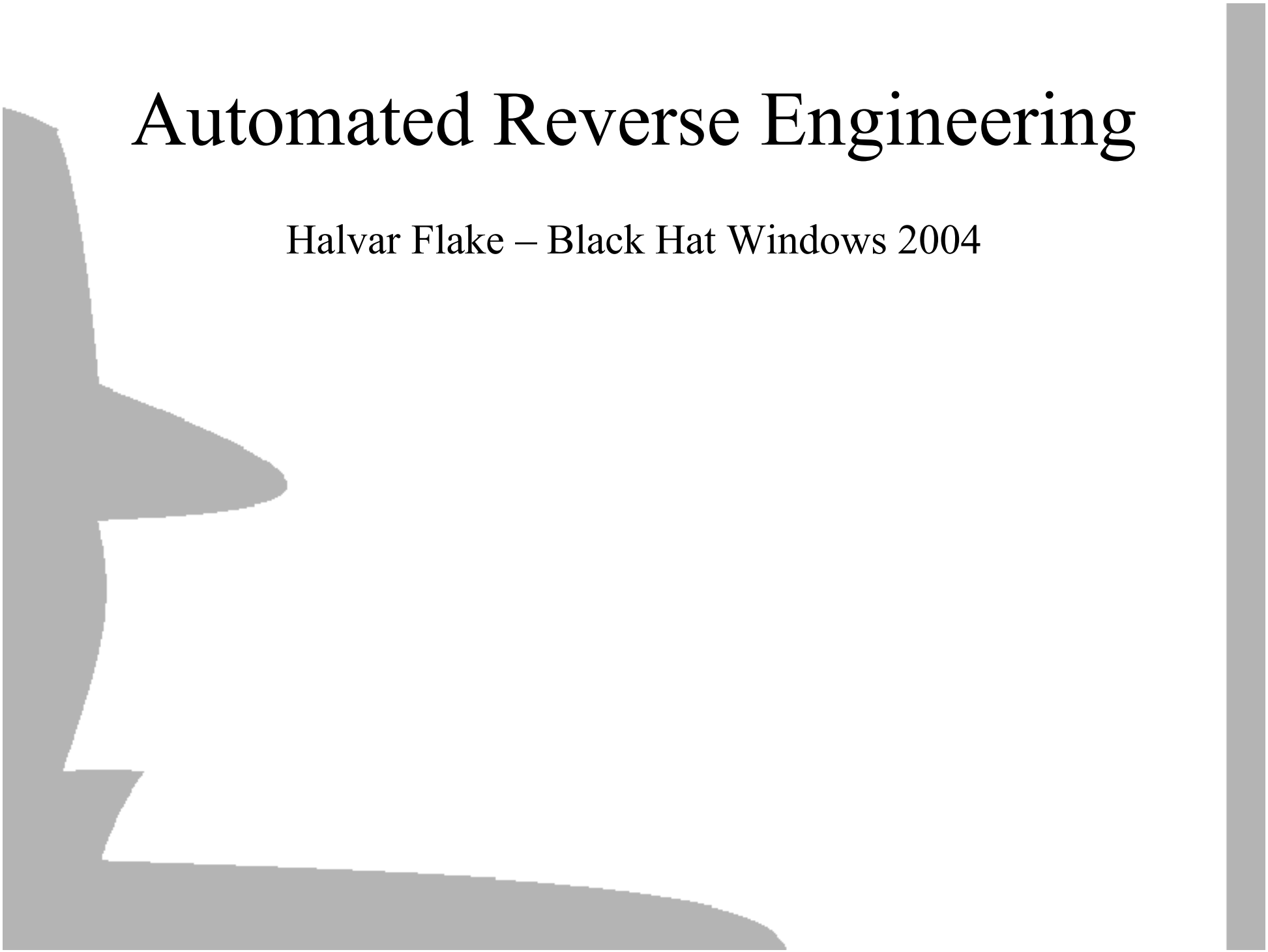


Automated Reverse Engineering

Halvar Flake – Black Hat Windows 2004

Outline for the talk (I)

Theoretical (Dry!) parts first, more “practical” in the second half

- Using simple IDC scripts to find security holes
- Recapitulation of the problems of “simple” analysis

- An intermediate assembly language
- Design considerations & details concerning automated translation

- Dataflow analysis on the intermediate language
  - Requirements
  - Feasibility
- Break (stock up on Coffee)

Outline for the talk (II)

"Practical" stuff in the second half

- Detecting Loops
  - Detecting memory writing loops
  - Detecting memory copying loops
  - Estimating the amount of iterations of a loop
- Binary DIFF
  - The importance of DIFF'ing code
  - Problems with binary DIFF'ing
  - Solution: Structural Differences
  - Example Dissection of Microsoft's Messenger and Workstation patches
  - Porting Symbolic Information between versions

Simple IDC scripts

- Published in 2000/2001
- Added HTML generator, released as “BugScam” (http://projects.sourceforge.net/bugscam)
- Very very very limited in scope
  - No dataflow analysis
  - No advanced code understanding
  - RATS/ITS4 for binaries ☹
- In conjunction with a structure reconstructor still occasionally useful
- Not very good for “in-depth” analysis
- Useful for “bulk-scanning” – 250 GB Harddisks are cheap, and IDA has a batch analysis mode

Simple IDC scripts (I)

- Will only flag bugs occuring from misuse of standard library functions
- Need hand-written analysis scripts
- Will generate a tremendous amount of false positives (90% of all alerts – not quite as good as IDS yet ☺)

➔ No matter how stupid an analysis tool is, some programmers will make mistakes which are “stupider”

➔ Demonstration: MS Media Server …

Problems of “simple” analysis

- Some bugs are very complex
- Simple analysis is CPU dependent
- Simple analysis does not understand “complex” situations
  -- but almost anything that involves passing a few variables around is “complex”
- Even “advanced” simple analysis does not deal with pointers and using pointers to access memory

➔ Tools that can be used to find complex bugs are complex to write

Dataflow Analysis

- Many bugs can be thought of as dataflow analysis problems
- “Can value X reach program point Z ?”

- Dataflow analysis is a well-trodden (and complex) academic field (check ISBN 3-540-65410-0 if you are interested ☺)

- I am naïve, I just write my own ™ ☺
- I do not want to write new code for every new CPU I want to support

➔ Intermediate Assembly Language / MetaCPU is needed

MetaCPU (I)

MetaCPU design decisions:
(Yipie ! I get to invent my own assembly !)

- RISC – like
- SPARC – like argument passing
- No practical restriction on numbers of registers:
  - 256 Global registers
  - 256 Temporary registers
  - 256 Local registers
  - 256 Input/Output registers
  - 256 Flags registers
  - PC, SP, FP

MetaCPU (II)

Why RISC – like ?

- Every instruction has to be handled by the analysis layer
- Fewer instructions == less code to write
- Fewer instructions == less complexity == fewer mistakes

Why SPARC – like argument passing ?

- Simplifies dataflow analysis – registers are “un-aliasable”, e.g. they cannot be accessed via pointers – stack passing would be more expensive
- SPARC is very very beautiful ☺

MetaCPU (III)

Why 256 Flags registers ?

- Branch condition analysis will be needed in the future
- Different CPU's make branching decisions differently
- Flags must be superset of flags on all CPU's that are supported
- Better too many than too few !

Why stack at all ? Why not convert all [esp+XX] accesses to further register access ?

- Local variables must be accessible via pointers !

MetaCPU (IV)

MetaCPU design decisions:

- No REP-like instruction
- No indexed/scaled addressing modes
- Further reduced RISC (RRISC ?)
- 3-operand architecture
- Explicit memory access (“LDM”, “STM”)
- No madness such as “direction flags”
- Explicit signedness of comparisons in branches (if possible)

MetaCPU (V)

Why no REP-like instructions ?

- Loops should be explicit in the flowgraph, not “hidden” !

Why no indexed/scaled addressing modes ?

- `mov eax, [ebp + ecx * 8 + 0x260 ]` is hard to analyze

Why further reduce RISC ?

- Who needs “INC” or “DEC” if you have “ADD” ?

Why explicit memory access (“LDM”, “STM”) ?

- Easy distinction of memory-modifying instructions from non-memory-modifying instructions (Benefits will become more clear in second half of talk)

MetaCPU (IV)

Why explicit branch signedness ?

- “br_sl” (Branch signed lower) is nicer to read than “jng” (Jump not greater)

MetaCPU Translation

Quote from the Hagakure (German translation)

“ Big problems should be approached freely,
small problems should be approached with great care “

Many small engineering problems had to be solved …

- Converting stack argument passing (and saving registers on the stack) to passing arguments in registers was tricky
- Dealing with weird (and changing) compiler optimization schemes took more time than intened

➔ Before: “ This is going to take about 1 month ! ”

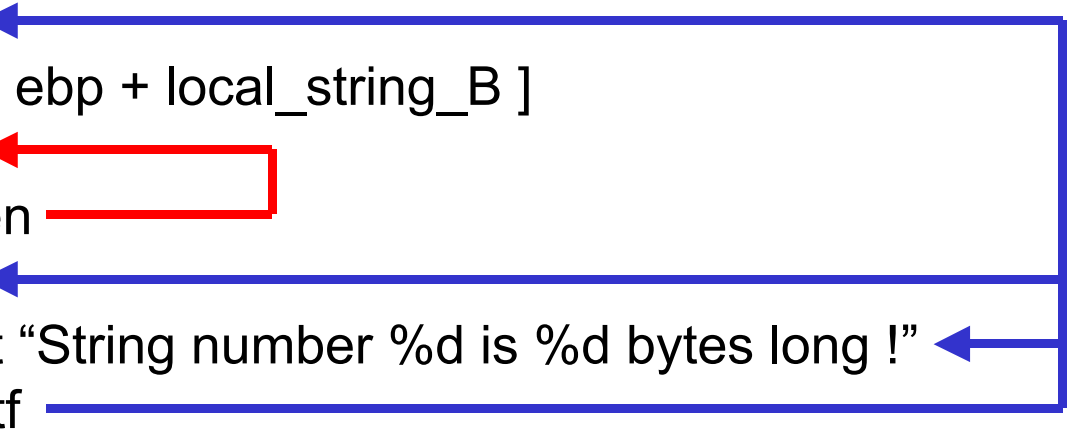
➔ Result: Many months

MetaCPU Translation

Stack-passing to argument passing:

- Number of arguments to EVERY call has to be known !
- Optimizing compilers will generate code like this:

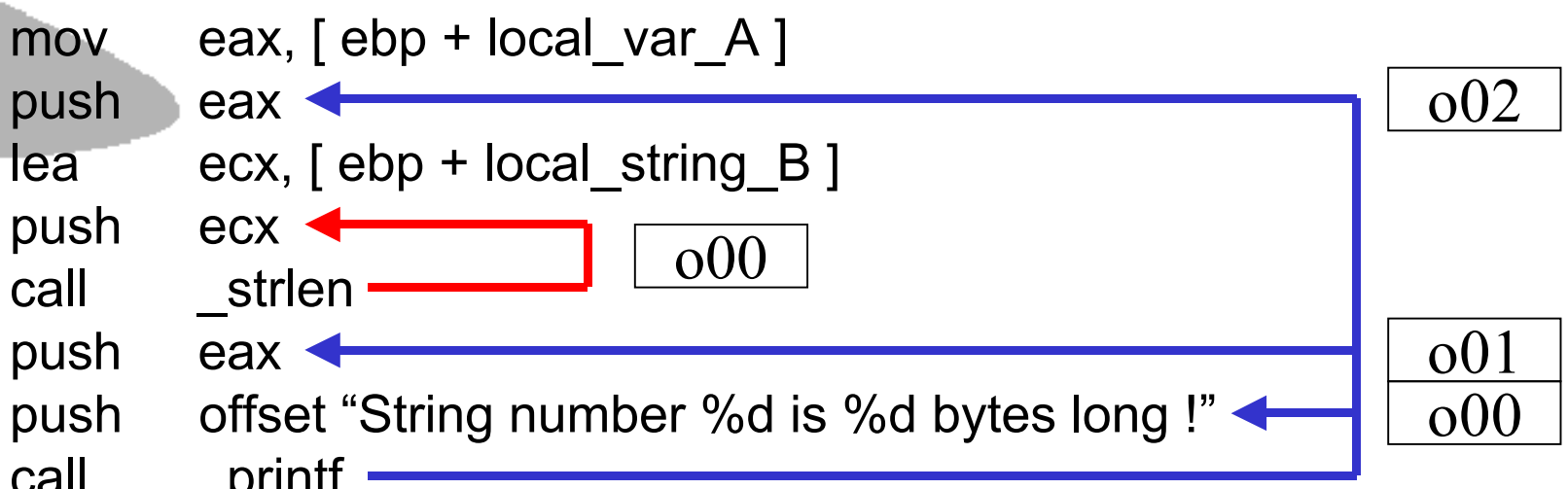
```
mov     eax, [ ebp + local_var_A ]
push    eax
lea     ecx, [ ebp + local_string_B ]
push    ecx
call    _strlen
push    eax
push    offset “String number %d is %d bytes long !”
call    _printf
```



MetaCPU Translation

Stack-passing to argument passing:

- Number of arguments to EVERY call has to be known !
- Optimizing compilers will generate code like this:



MetaCPU Translation

Stack-passing to argument passing:

- Number of arguments to EVERY call has to be known !
- How to deal with variable argument functions ?
  [… printf(), vsnprintf() …]

- "Cheating": User-supplied annotations in the form of
  <CALL_ARGS> XX </CALL_ARGS>

(Demonstration of mistranslation without proper annotations)

Compiler optimization

Other challenges: Modern compilers do strange optimizations:

→ Replacing push with mov:

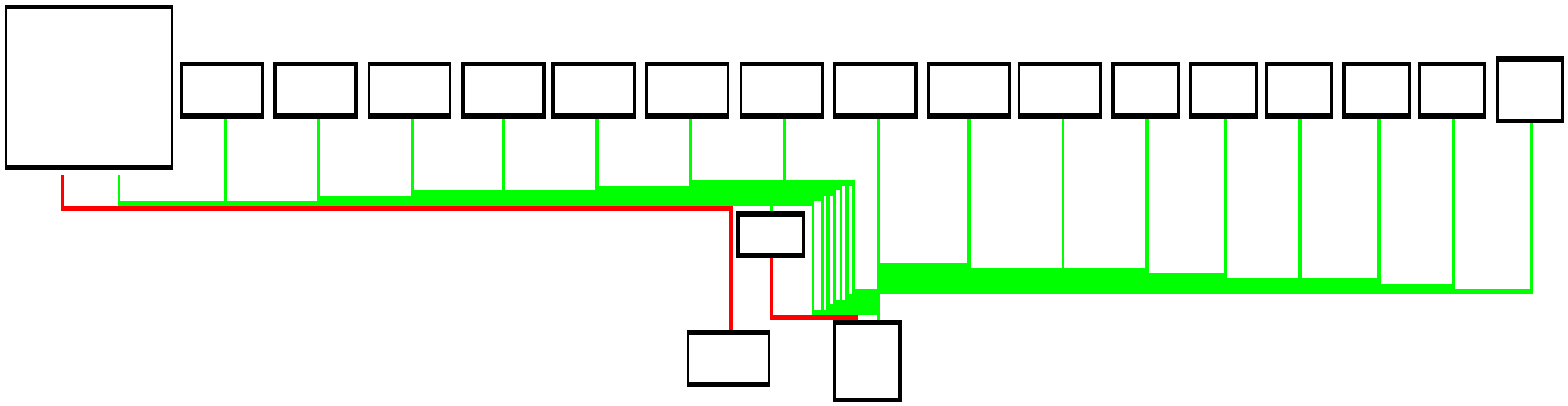
Old:
```
push    eax
push    ecx
call    function
add     esp, 8
```

New:
```
sub     esp, 8
mov     [esp+0], ecx
mov     [esp+4], eax
call    function
```

→ Translation from "mov [esp + offset]" to the appropriate output register is needed as well …

Compiler optimization

Other challenges: Modern compilers do strange optimizations:
Merging of exit blocks is getting fashionable again…



Compiler optimization

Examples for compiler madness:

→ Merging exit blocks of functions

- Watcom C++ did this in the old days
- Everybody thought it was gone for good
- Attempt at using less memory (and thus improve cache performance on P3's)
- Only identical parts of functions can be merged
- Functions have to be identical from point of merger on !

→ Only epilogues (e.g. the last 1-2 blocks) are normally merged

→ Facilities for seperating merged functions are needed

Translation notes

- Translation cannot be proven to be correct
- In order to prove correctness of translation, a one-to-one mapping ("bijective") between CPU states & instructions would need to exists
- As MCPU is supposed to represent multiple architectures, It would have to become a superset of all CPU's

→ MCPU would then be accumulation of all bad ideas and all complexity of all CPU's ➔ Useless

(My lame excuse for not getting the translation to be provably correct)

→ We have to "trust" the translation to be correct

Automated verification

- An emulator for the MetaCPU code would be useful
- Could be used to automatically test the correctness of the translation

- Does anyone in the audience know how I could clone myself a few times to get more work done ?

- Translation adds redundancy
- Peekhole optimizer would be useful to make the code more readable

… so much to do, and the days keep getting shorter …

Translation Example

```
0040124C:
push    ebp
mov     ebp, esp
push    offset 00403038aHelloWorld; " Hello World "
call    00401284printf
add     esp, 4
pop     ebp
retn
```

```
0040124c:      str          fp,      ---,      t00
0040124d:      str          sp,      ---,       fp
0040124f:      str    00403038,      ---,      o00
00401254:   invoke    00401284,      ---,      ---
0040125c:      str         t00,      ---,       fp
0040125d:   return         ---,      ---,      ---
```

Translation Example (II)

```
00401329:
mov     eax, [ebp+arg_0]
mov     ecx, 0047F050dword_47F050
mov     eax, [ecx+eax*4]
push    eax                     ; hObject
call    ds:00499688CloseHandle
mov     eax, [ebp+arg_0]
mov     ecx, 0047F050dword_47F050
mov     dword ptr [ecx+eax*4], 0
push    0
call    0046F6C0x1t___endthreadex
add     esp, 4
jmp     00401360loc_401360
```

Translation Example (III)

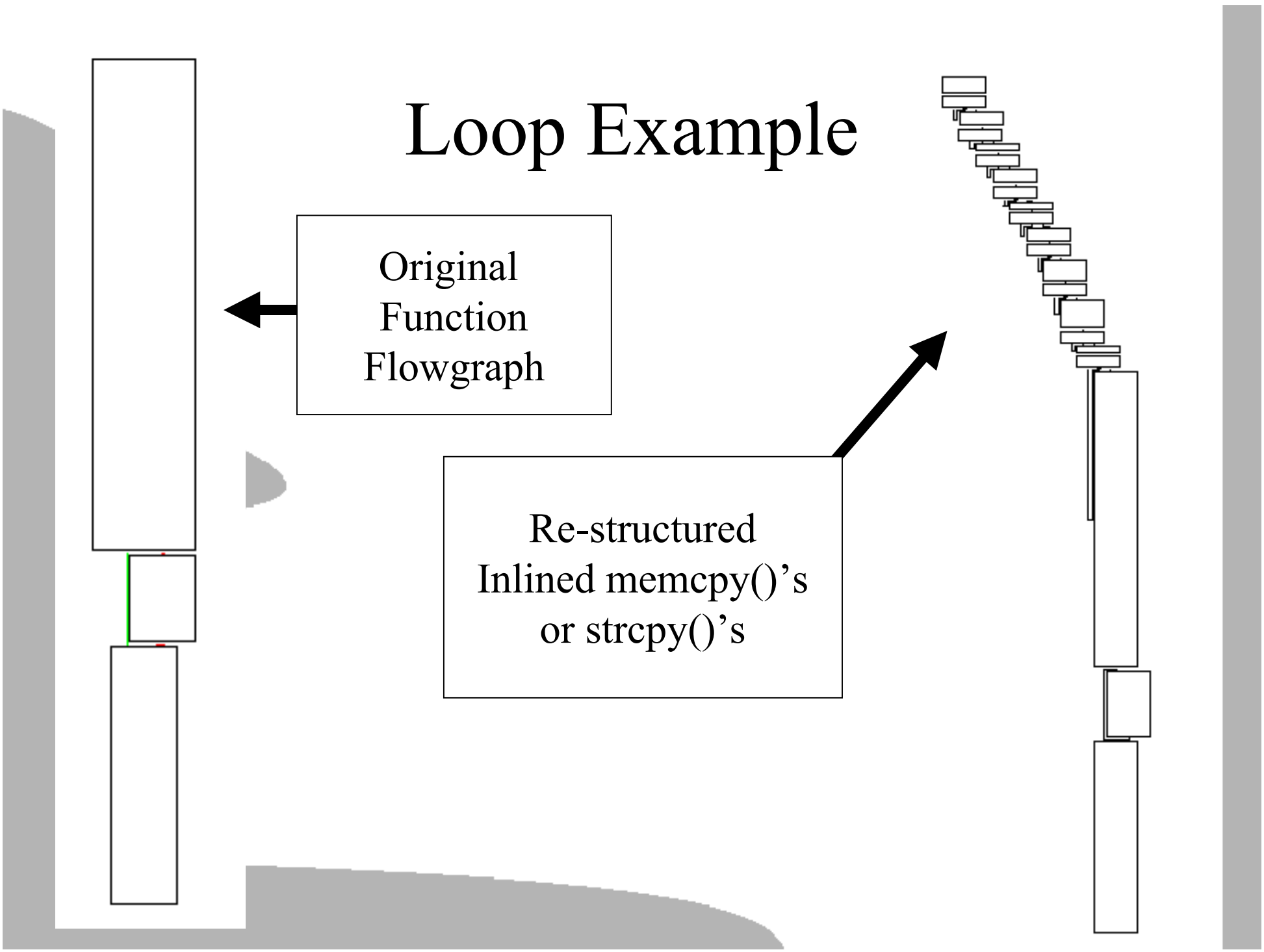
```
00401329:      str        i00,       ---,       g00
0040132c:      ldm   0047f050,       ---,       g01
00401332:     imul        g00, 00000004,       t04
00401332:      add        g01,       t04,       t04
00401332:      ldm        t04,       ---,       g00
00401335:      str        g00,       ---,       o00
00401336:      ldm   00499688,       ---,       t00
00401336:   invoke        t00,       ---,       ---
0040133c:      str        i00,       ---,       g00
0040133f:      ldm   0047f050,       ---,       g01
00401345:     imul        g00, 00000004,       t04
00401345:      add        g01,       t04,       t04
00401345:      stm   00000000,       ---,       t04
0040134c:      str   00000000,       ---,       o00
0040134e:   invoke   0046f6c0,       ---,       ---
00401356:   branch   00401360,       ---,       ---
```

REP → Loop

- The abstract CPU does not have a REP instruction prefix
- Translation of REP's through an artificial loop & control flow restructuring
- Expands code length significantly
- Clarifies true extent of "loopyness" a function has

→ Benefits will become clear in the second half
→ Can be used to automatically detect inlined strcpy's
→ Can be used to automatically detect inlined memcpy's

Loop Example



Data Flow analysis

As mentioned before:

- Many code analysis problems can be boiled down to Dataflow analysis problems
- Asking questions such as

  - What values can register X have at program point Y ?
  - What values are being calculated that have register X as input variable ?

- Complex problem (still working on a solution ☺)
- General cases have been shown to be unsolvable – but specific cases in particular practical programs ?

Data Flow Requirements

For security analysis, our data flow needs to …

- Be interprocedural – many security bugs have their effect in a different function than their cause !
- Be able to deal with aliasing of memory locations, passing arguments by passing a pointer to a structure etc.
- Be context-sensitive (e.g. needs to know where it was called from)
- Be (if possible) provably complete (so that we can say we're certain there is no heap-corruption problem)

➔ By common consensus, this is impossible ☺ (in the general case

➔ But we do not care about the "general case", we have very specific practical cases

Data Flow analysis

Very dry subject, many open questions:

- ObjRec includes a (simple) dataflow engine
- Sobek includes a (simple) dataflow engine

- Simple engines track “which register gets stored in which register” or “which register gets stored in which local variable”

- Some useful information can be gained, but for serious analysis more complex constructs have to be handled

Data Flow analysis

Passing arguments by pointers/references:

```
int maxlen = 20;
char buf[ 20 ];
char stuff = buf;

If( fill_buf( stuff, &maxlen ) == 0)
{
    stuff = malloc( maxlen );
    fill_buf( stuff, &maxlen );
}
```

Looks simple to handle … but …

Data Flow analysis

Passing arguments by pointers/references:

```
struct a { int a; int b };
char buf[ 20 ];
char stuff = buf;
a.b = 20;
If( fill_buf( stuff, &a ) == 0)
{
    stuff = malloc( a.b );
    fill_buf( stuff, &a );
}
```

Data Flow analysis

- We have to work on the disassembly, so no type info available
- We cannot easily distinguish between
  struct a { int a; int b; } and
  int a; int b;
  For us, both will look identical !

Indirection in becomes a problem:

How do we represent multiple layers of indirection ?
  local_var→memberA→memberAA→memberAAA

Representing indirection

- Every non-global variable in a program can be represented in the format of

  Base Register + offset1 + offset2 … etc

- Every "+" denotes a memory dereference
- Arbitrary levels of indirection can be represented this way
- Structures can be represented this way
- Pointers to pointers to pointers to pointers to pointers can be easily represented, too

Representing indirection (II)

Examples:

- structure member (g00 holds pointer to structure)
  g00 + offset_member
- local variable:
  fp + offset_var
- structure member (local variable holds pointer to structure)
  fp + offset_var + offset_member
- structure member “A” in some structure which is pointed to by a pointer in structure ”B” whose pointer is stored in a local variable
- fp + offset_var + offset_B + offset_A

Indirection Example

```
strucA = malloc( sizeof( STRUCTA ));
strucB = malloc( sizeof( STRUCTB ));

strucA->member10 = var_to_track;
strucB->member20 = strucA;
```

Assuming strucB is stored as a stack variable at FP + 30, the representation of var_to_track is now:

FP + 30 + 20 + 10

Aliasing

Another issue is aliasing – if we track a memory location, what other parts of the program have a pointer to it and can thus access it ?

- Another complex problem
- Can sometimes be solved solved by taking into account restrictions imposed by the C language

- Anyone still awake ? ☺

Dataflow Summary

- Really hard problem
- Many different approaches
- No code done that I am satisfied with yet
- Once complete dataflow is accessible, many security problems become easy to detect

➔ Building provably complete dataflow is my favourite problem nowadays

➔ A professor from NUS is responsible for this: He told me to “not settle for heuristics” but try to do the “real thing”

➔ I don’t know if I’ll ever get it done ☺

➔ Even imperfect dataflow reveals interesting information (Demonstration)

Dataflow Summary (II)

- Hopefully next year I will be able to show a fully working program ☺
- If I never manage to get it running, I will at least have learnt a lot along the way
- The second half of the talk will be about a few practical things that I "found" on the "side of the road" during my "quest" for dataflow analysis

- Coffee, anyone ?

➔ BREAK

Second Half

"Practical" stuff in the second half

- Detecting Loops
  - Detecting memory writing loops
  - Detecting memory copying loops
  - Estimating the amount of iterations of a loop
- Binary DIFF
  - The importance of DIFF'ing code
  - Problems with binary DIFF'ing
  - Solution: Structural Differences
  - Example Dissection of Microsoft's Messenger and Workstation patches
  - Porting Symbolic Information between versions

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 ?

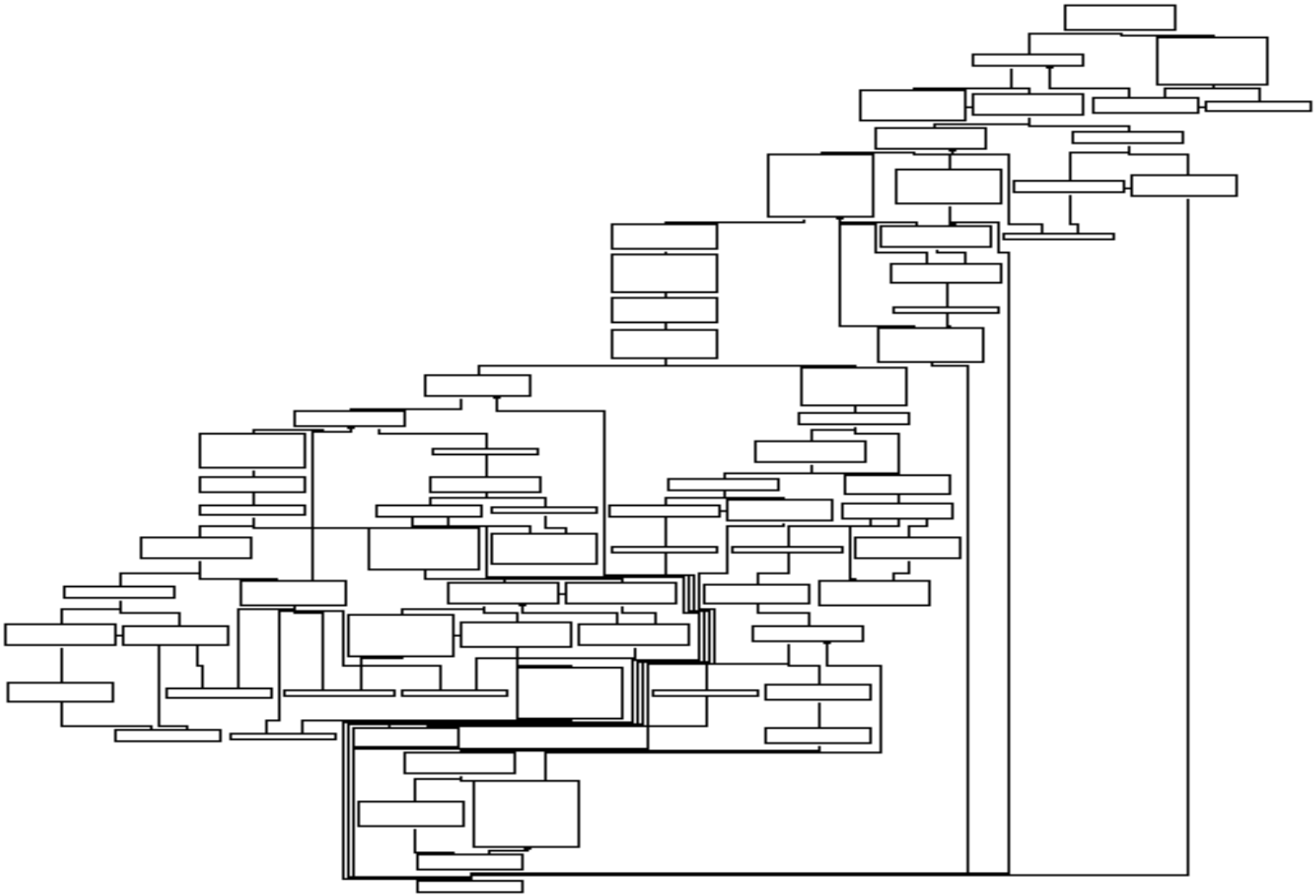
→ Memory – copying loops (decoding etc) seem to have been neglected

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

→ A mechanism to spot loops in binaries is useful

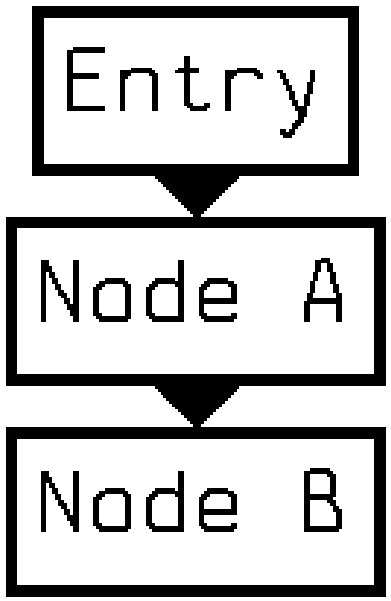
Loop detection (II)

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*

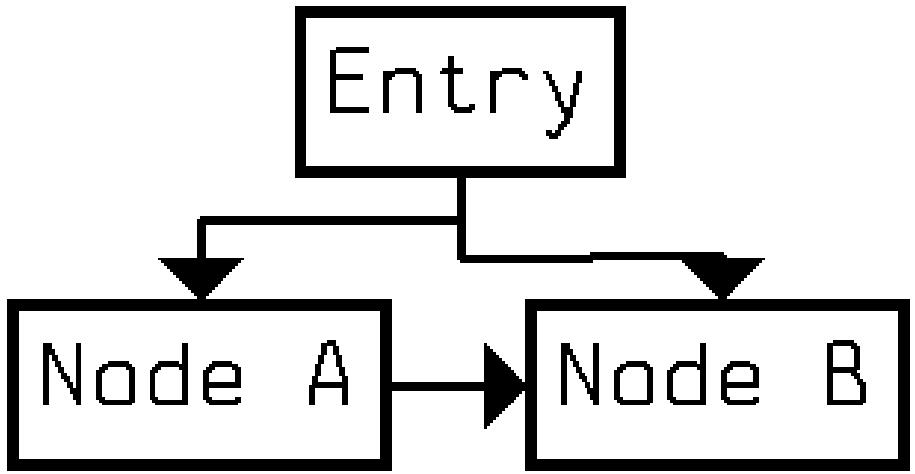


B is dominated by *Entry* and also by *A*

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)

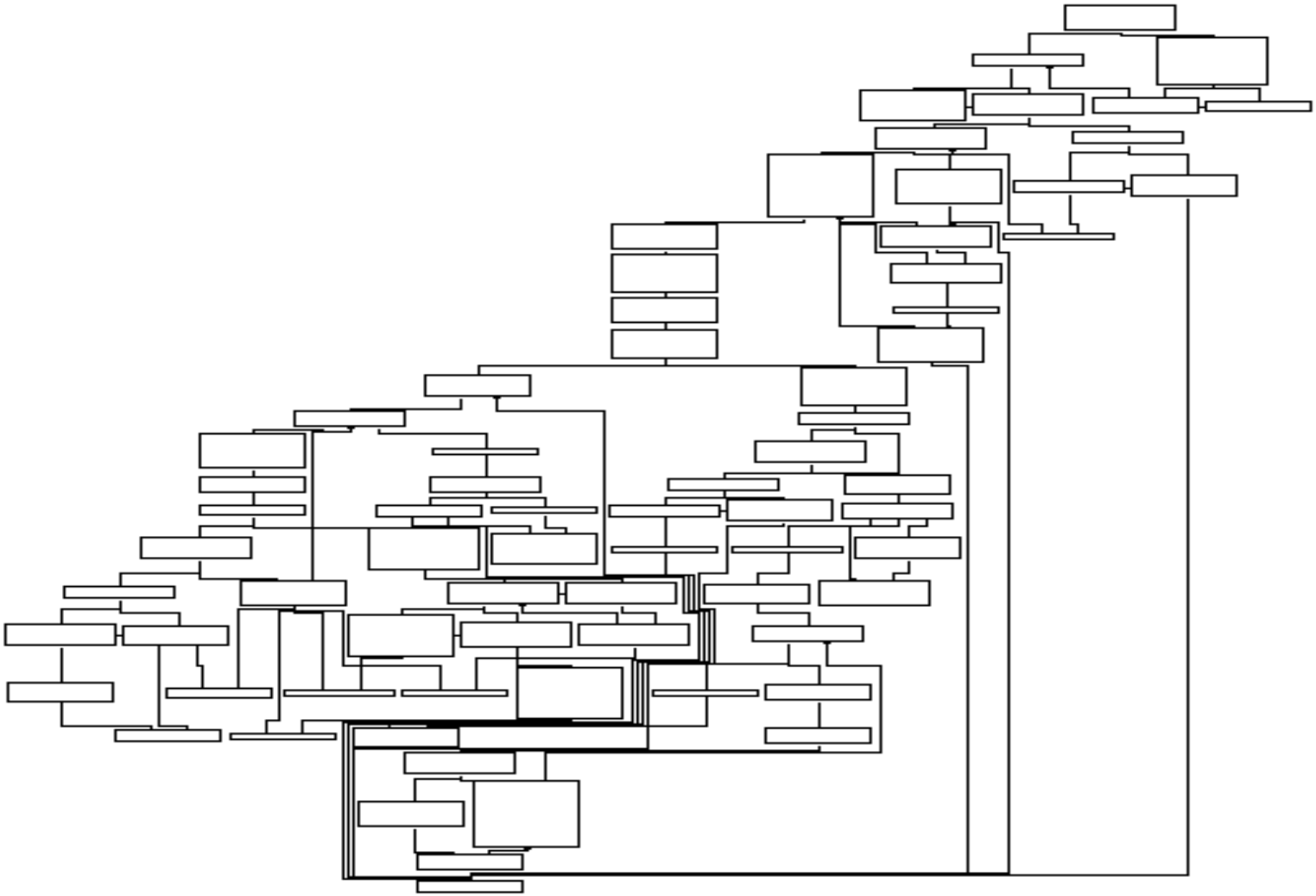
- Dominator Trees can be used to detect loops in graphs
- If a node *A* links to a node *B*, and if *B* dominates *A*, the link closes a loop in the graph

→ 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

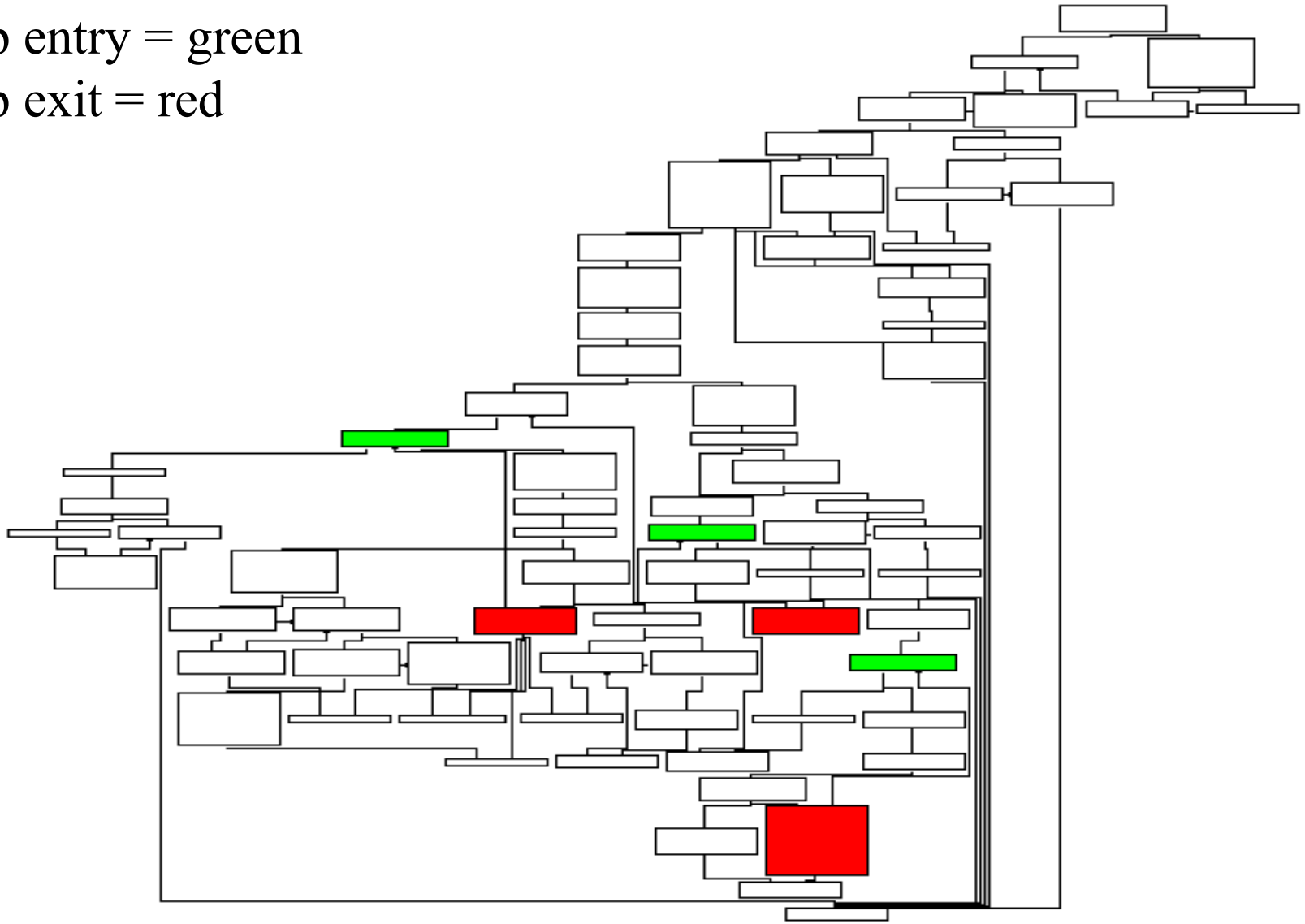
Loop detection (IV)

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 to the same location on every iteration)
- Loops that write a statically defined number of bytes are not terribly interesting

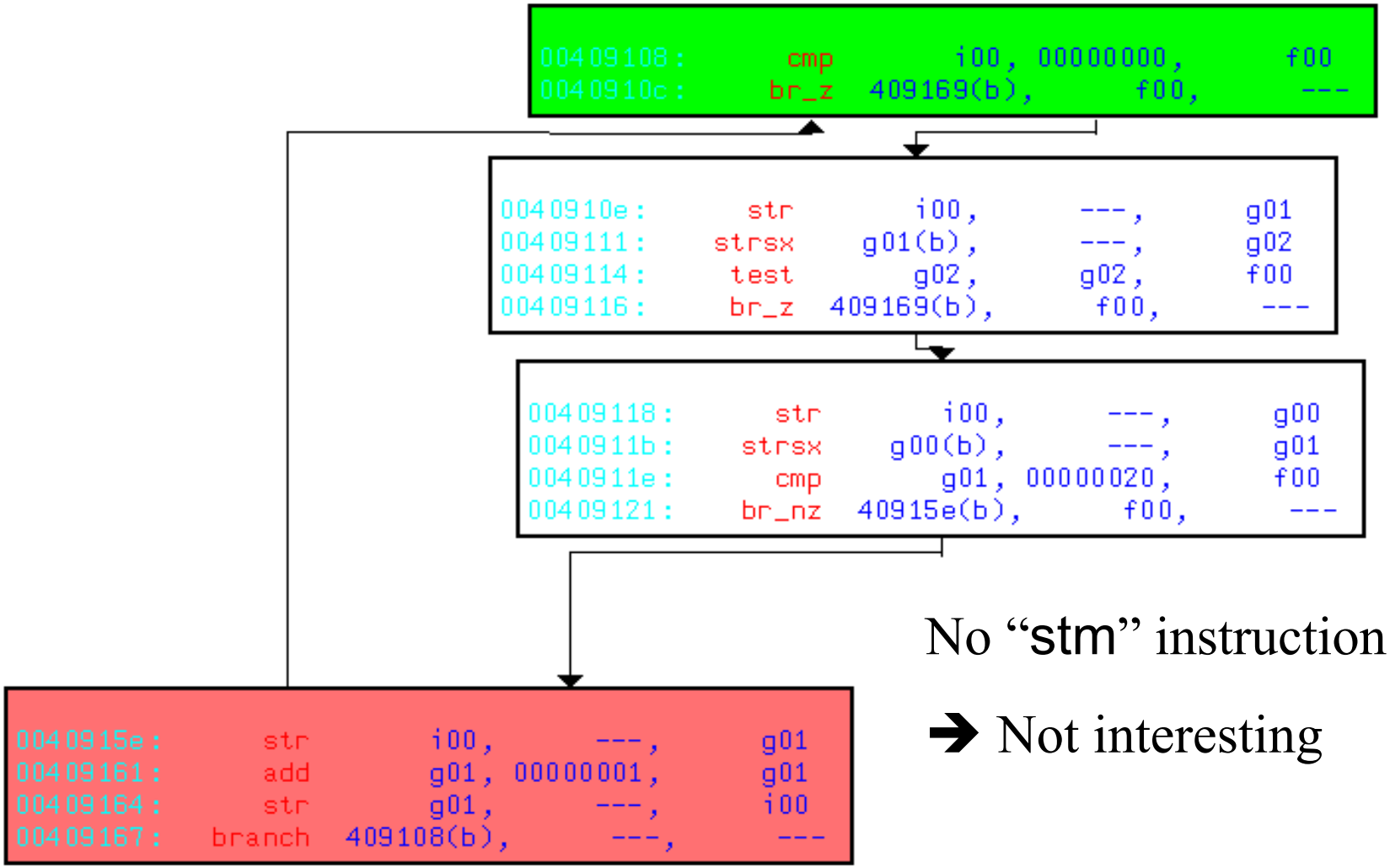
→ We want to eliminate all loops that do not write memory
→ We want to eliminate all loops that write to the same location on every iteration
→ We want to eliminate all loops that write a statically defined number of bytes

Memory-Writing

- The examined code has been translated to the MCPU code presented in the last talks (Blackhat Europe)
- All memory access is explicit, e.g. there is an explicit instruction for storing memory
- All implicit loops (repz movsd etc.) have been converted to true loops in the graph → inlined strcpy's and memcpy's are detected as well

→ 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

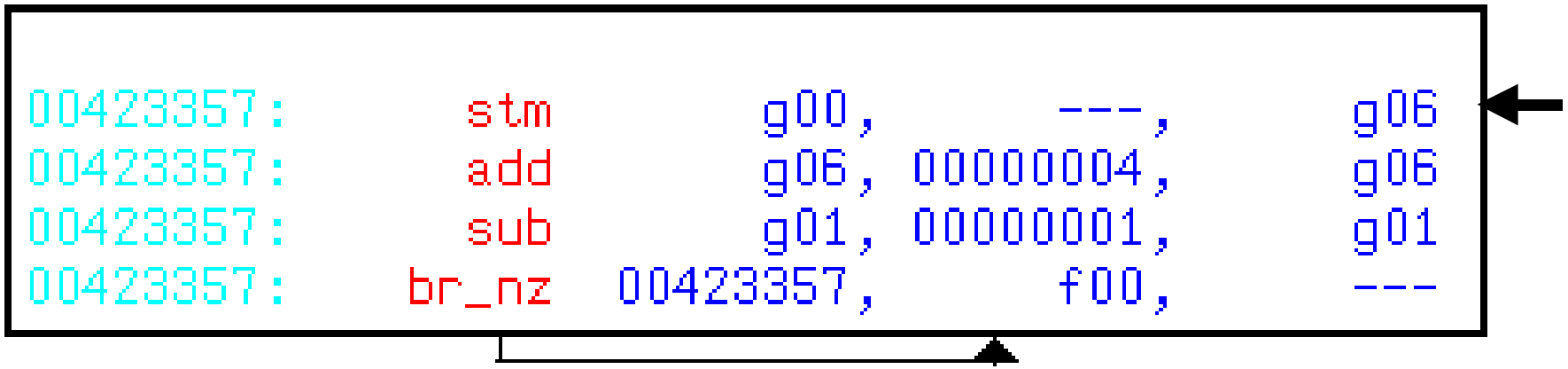
Variable-Writing

- A write access occurs in our loop
- If on every loop iteration, the location written 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)

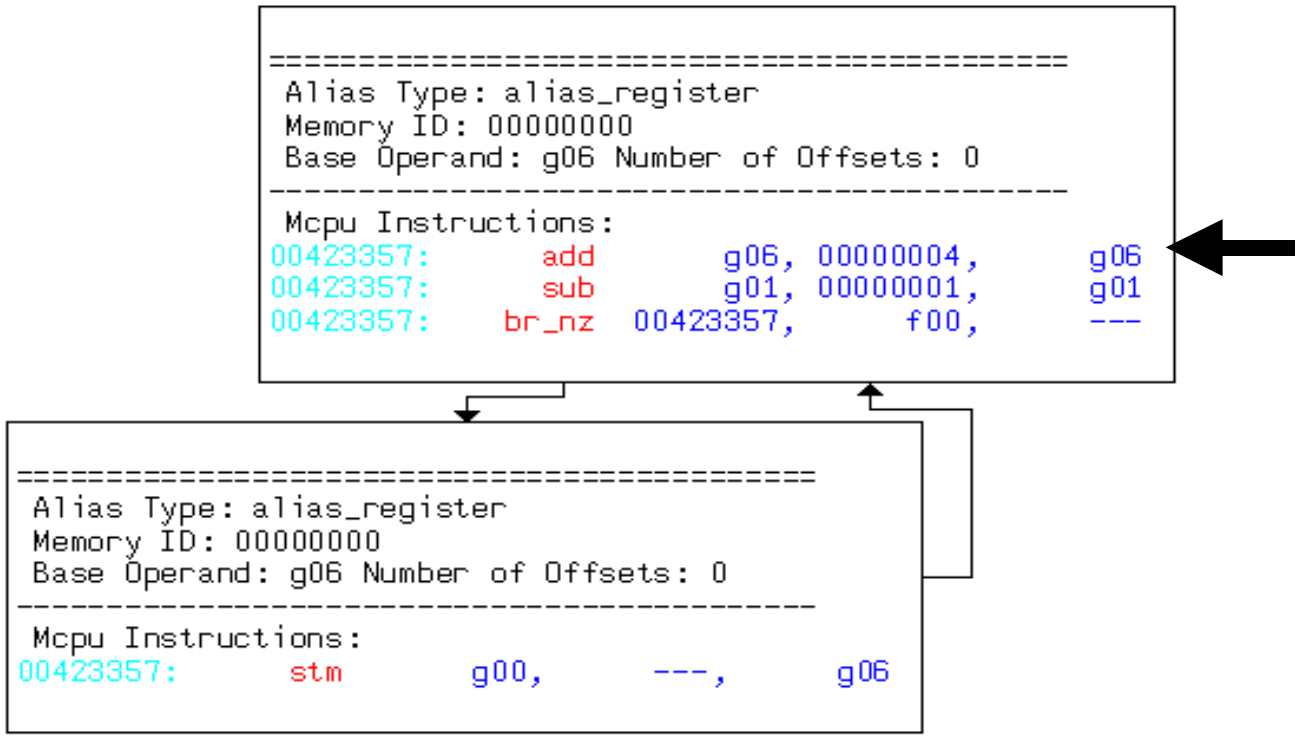
Considering a (very) simple example loop:



One memory store to the location pointed to by g06 occurs

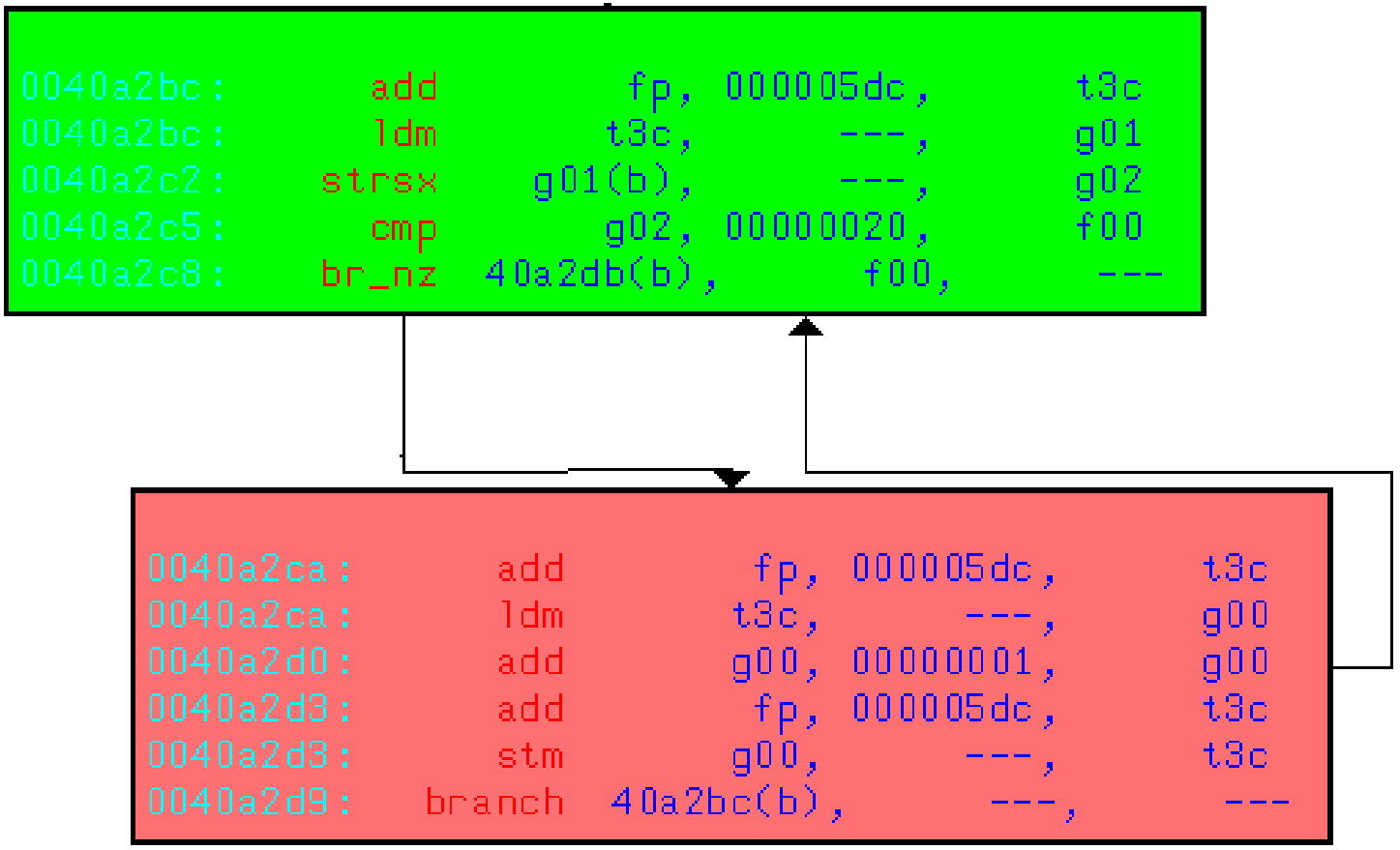
Variable-Writing (III)

A dataflow graph for this register is generated:



We can read from the graph that the pointer which is written to is incremented on every loop iteration ➔ dynamic !

Variable-Writing (IV)



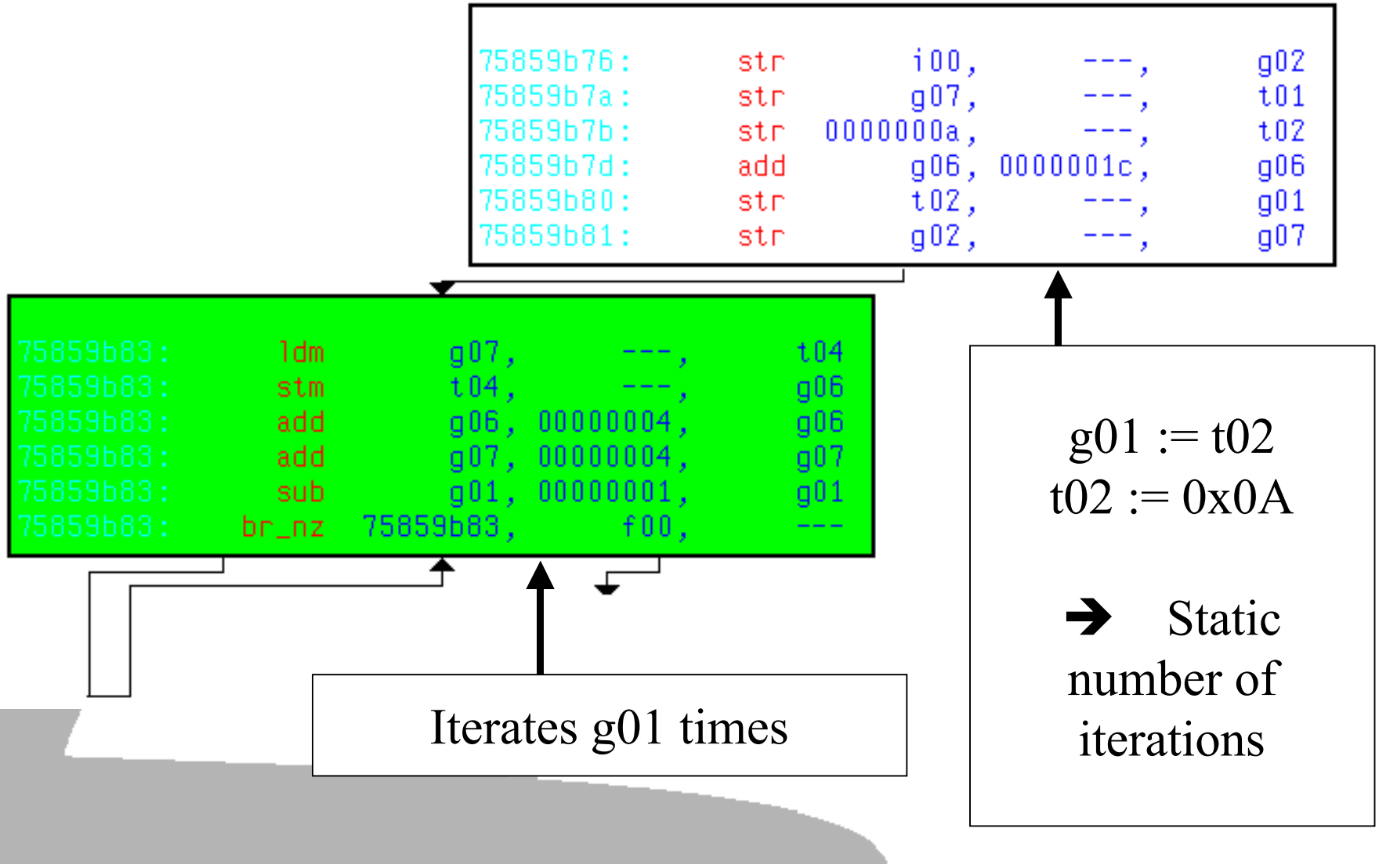
This loop is not interesting → loop_dfl_demo.vcg

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

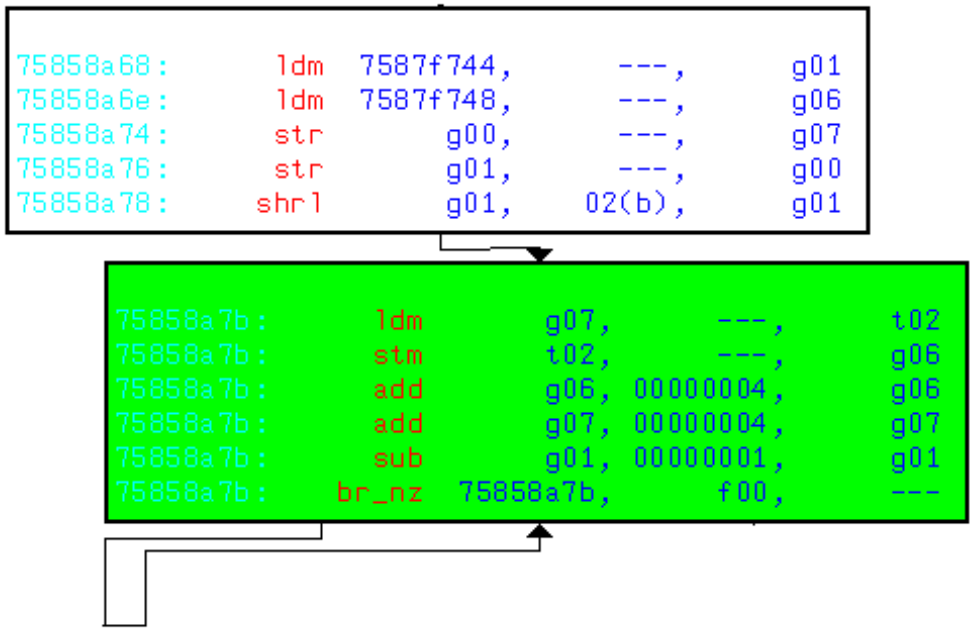
→ By eliminating all loops that iterate a predefined/static number of times, we can eliminate all loops that copy a static number of bytes

Defined Iterations (II)



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 (few, perhabs slightly more) 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 ?

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 (M. Zalewski)

Problems

- Normal pattern matching is problematic

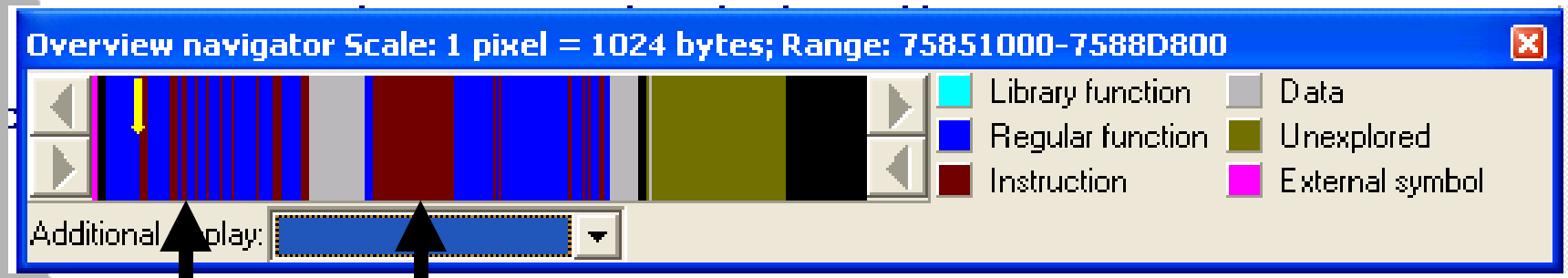
  → A few lines of code that change can lead to different register allocation and thus to many changed locations

  → 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

MS Internal Optimization

- Less-trodden path is moved to other pages
- Improves Paging and Cache utilisation



Less-trodden parts of functions

Heavily used parts of functions

Solution ?

- Structural fingerprinting ?

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

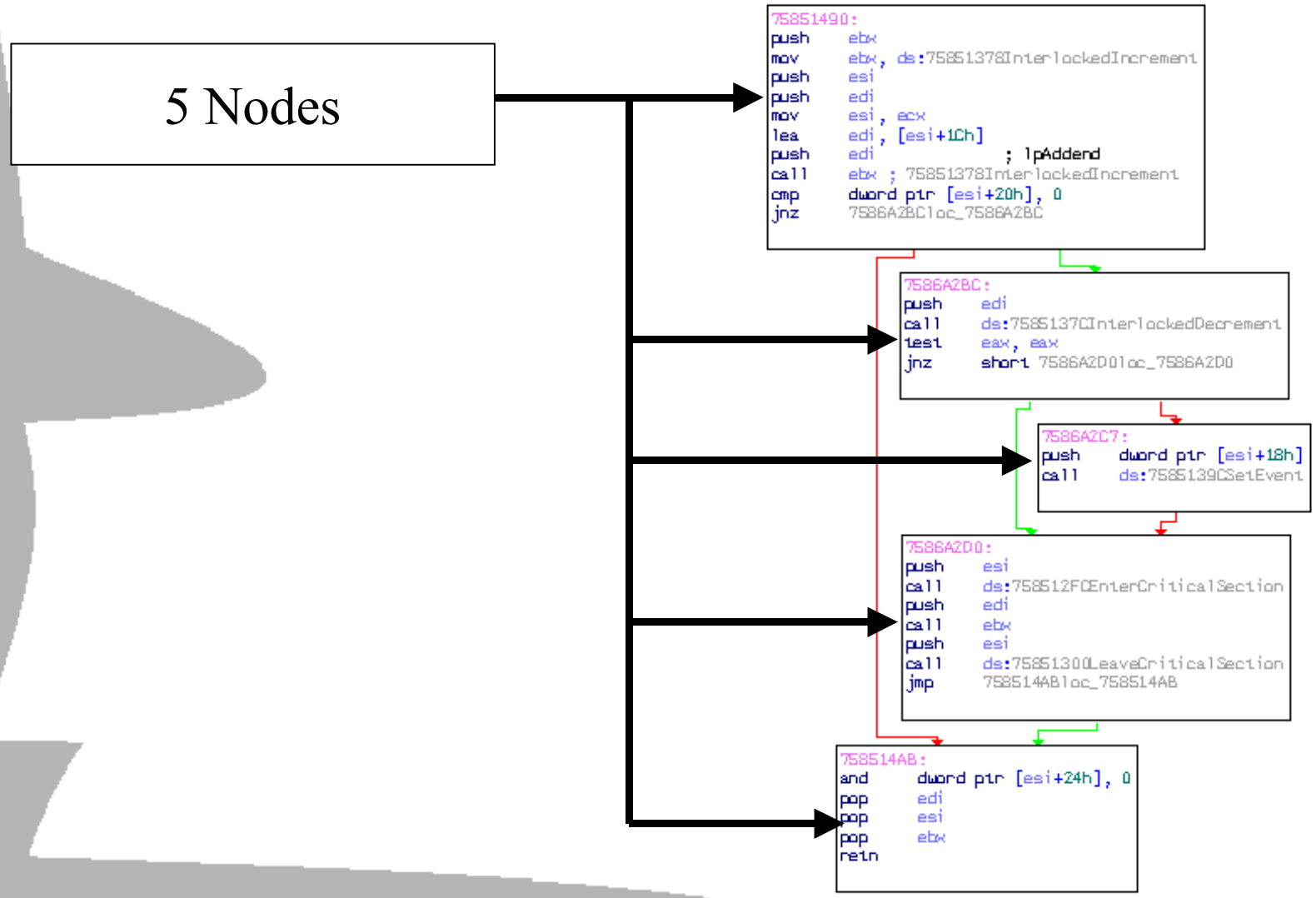
- Graph Isomorphisms (math-speak for finding out if two graphs are the same) are computationally expensive to compute

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

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

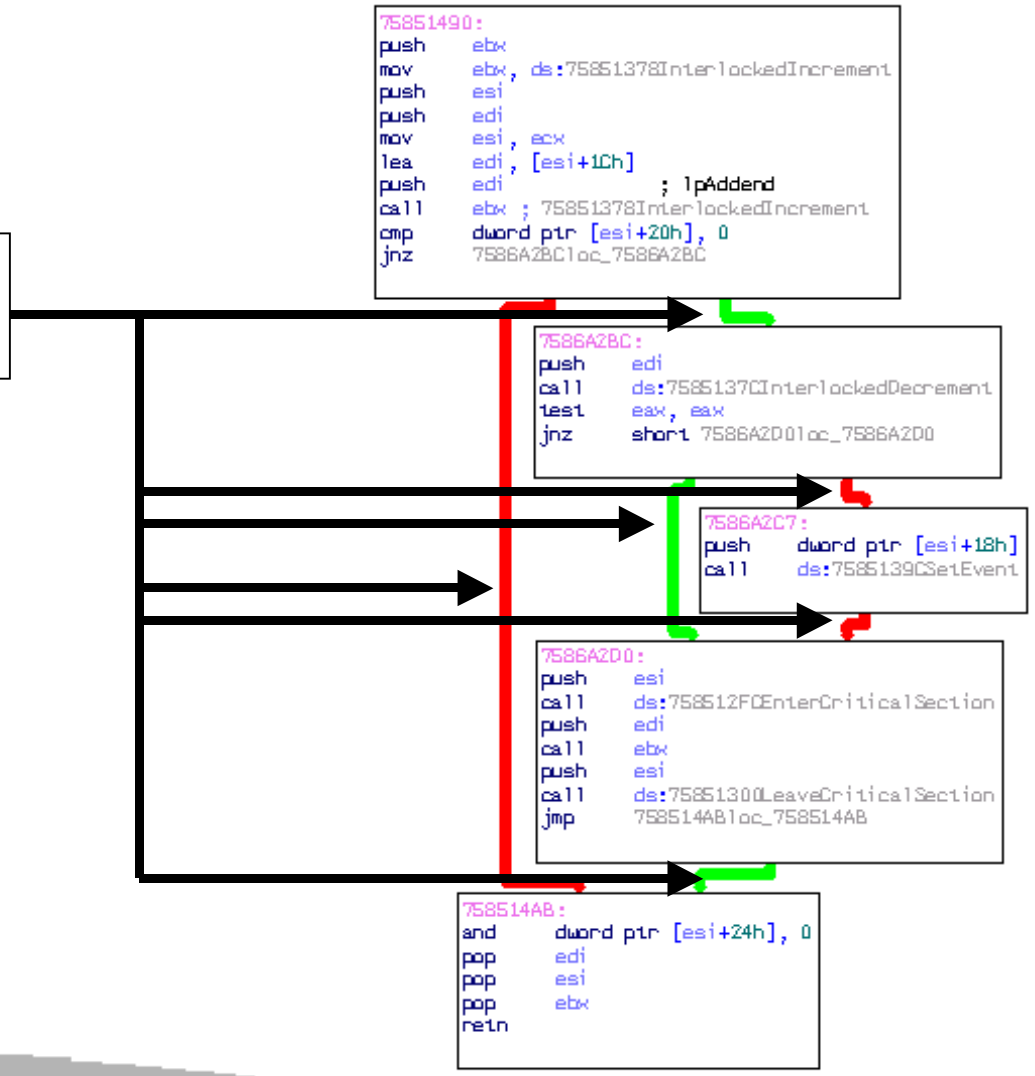
Example (I)

5 Nodes



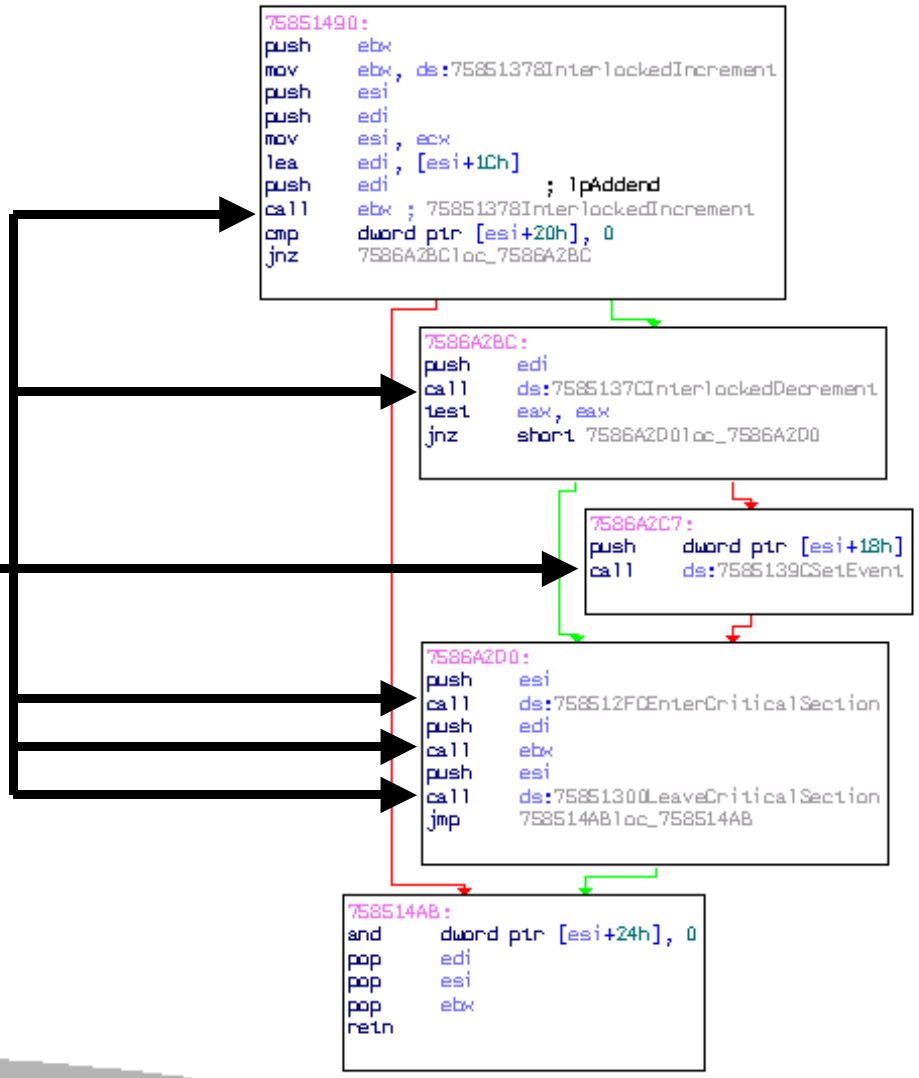
Example (II)

6 Links



Example (III)

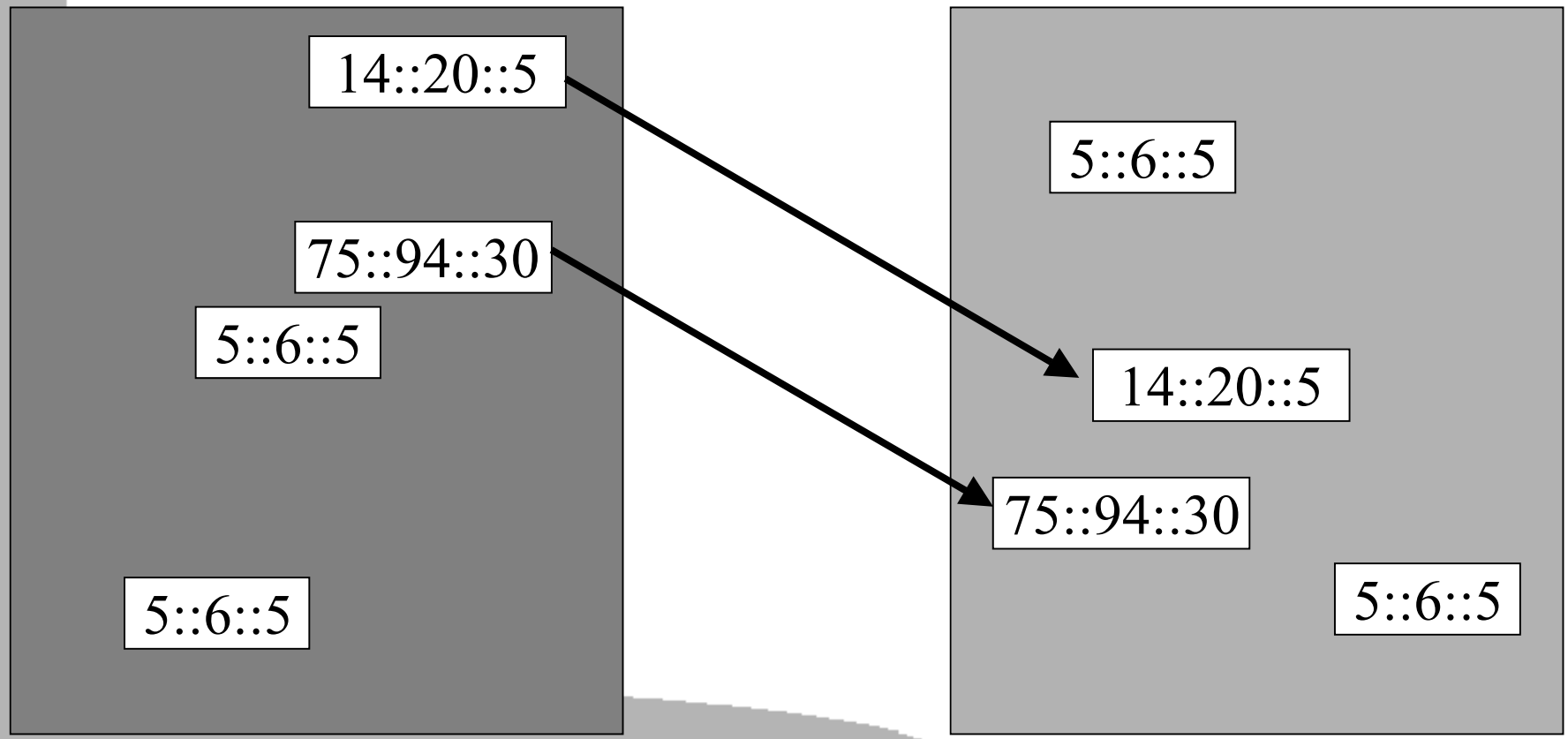
6 subcalls



Signature: 5::6::6

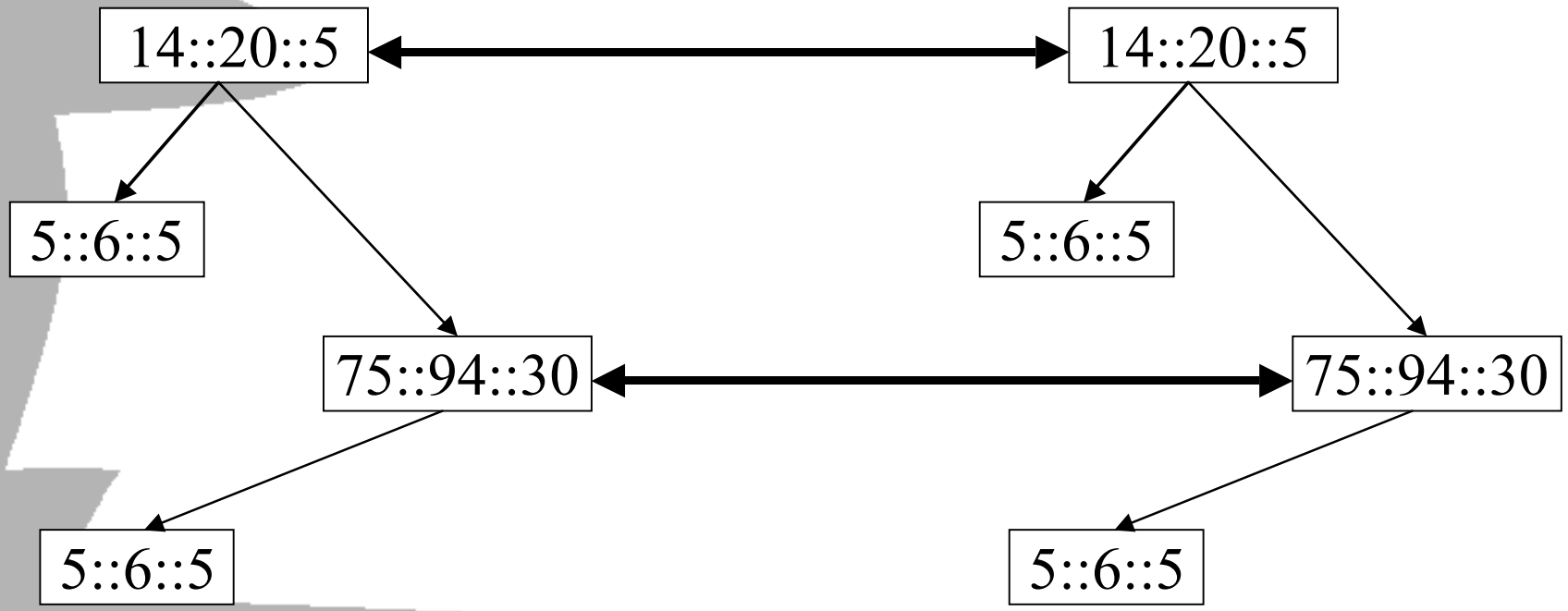
Fixedpoints

- All signatures from binary A are generated
- All signatures from binary B are generated
- Clear and unique mappings (fixedpoints) are detected



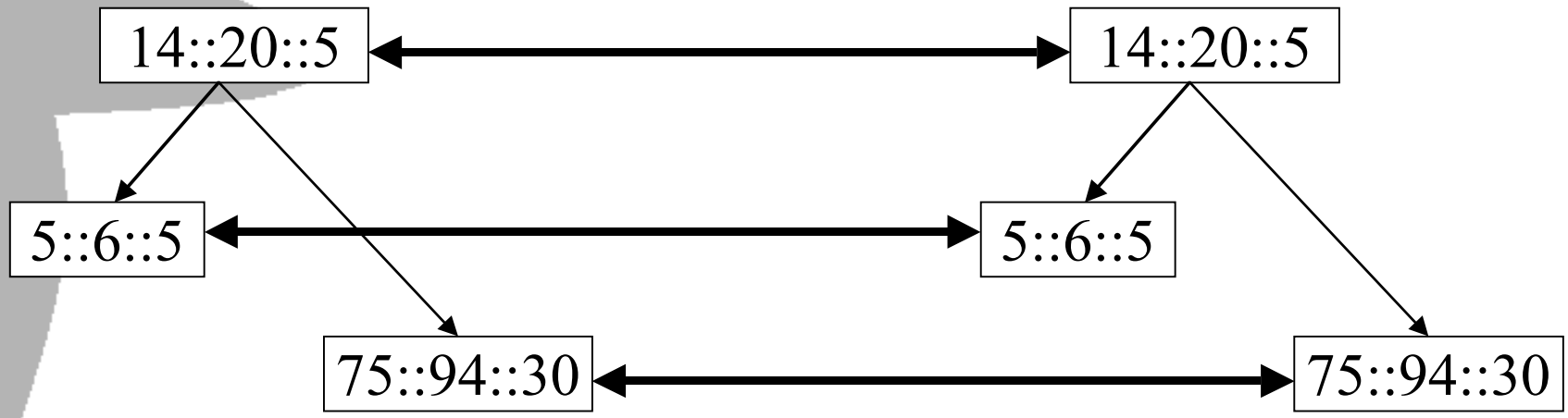
Fixedpoints (II)

- If multiple functions have the same signature, no useful match can be found
- A calltree for both executables is generated and used for additional matching



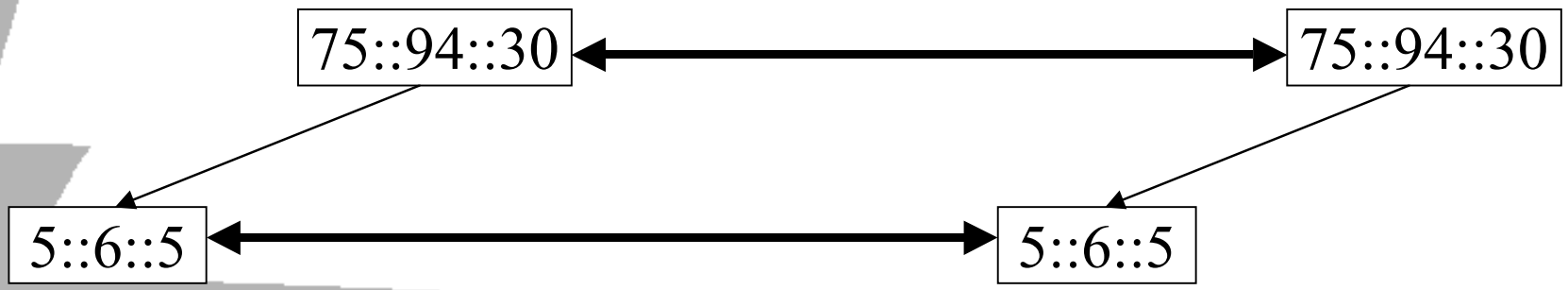
Fixedpoints (III)

- Instead of matching all signatures in A to all in B, all children of a particular fixedpoint in A are matched to the children of the corresponding function in B



Fixedpoints (IV)

- If multiple functions have the same signature, no useful match can be found
- A calltree for both executables is generated and used for additional matching



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 simple function)
- Simple functions can not be properly fingerprinted (but then again, do not change much either)

Porting old debug info

- Microsoft gives out debug information, but not for hotfixes
- Checkpoint and other vendors occasionally forget to strip their binaries

➔ Porting function names from older binaries is really useful

Demonstration:

Original File:

MSGSVC.DLL Win2k SP0 with Debug Info

New File:

MSGSVC.DLL Win2k after the overflow patched

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 cause are rare

→ Keeping bug information quiet after publication of an open source 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 → unpatched to patched diffs)

Questions ?

Thank you ! ☺