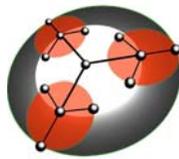


Compare, Port, Navigate

Three applications of graphs and graphing for security

Halvar Flake – Blackhat Briefings Europe 2005

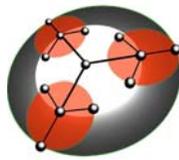


Overview

The talk consists of three parts:

- i. Comparing executable objects
 - i. “What is the difference between these two pieces of code ?”
 - ii. “How similar are these two pieces of code ?”
- ii. Porting information between executable objects
 - i. “If this malware new thing is similar to xyz, can’t I use my old disassembly ?”
 - ii. “They use OpenSSL in this embedded device, can I get those symbols into IDA ?”
- iii. Navigating executable objects
 - i. “This binary is huge – and where the heck should I start reading ?”
 - ii. “Now, how does all this fit together ?”

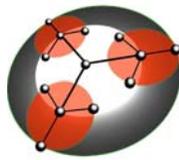
Structural Comparison



Introduction

- i. Security Patch Analysis became interesting to the mainstream with the first LSD DCOM bug
 - i. Special situation: Existence of critical security problem public, but no details available
 - ii. Few organisations patch in the first 10-14 days
 - i. Window of exposure between publication of security problem and fixing
 - iii. “Nonpublication of exploit details buys the customer extra time, because reverse engineering of security updates is hard”

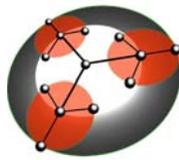
Structural Comparison



Asymmetry between source recompilation and RE

- i. Making a minor change and recompiling a program is easy
- ii. The general assumption is that reverse engineering these changes is hard
 - i. Reverse Engineer has to re-do all work, because he lost all results from previous disassembly
 - ii. Reverse Engineer then has to compare the function's logic, as many instructions will have changed due to optimisation
- iii. Software Vendors and HLL-Virus Authors try to exploit this asymetry
 - i. The first want to buy time for customers
 - ii. The second can create variants quickly/easily

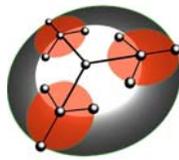
Structural Comparison



Diff'ing executables is difficult

- i. Small changes in the source code can trigger significant changes in the executable:
 - i. Adding a structure member will change immediate offsets for all accesses to structure members after the change
 - ii. Adding a few lines of code can produce radically different register assignments and lead to differing instructions
 - iii. Changed sizes of basic blocks in one function can lead to code in unrelated functions changing (because of branch inversion)
- ii. The overwhelming majority of changes in the binary are irrelevant
 - i. Classical trade-off: More false positives or running the risk of a false negative ?

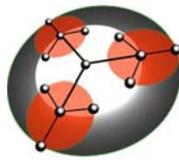
Structural Comparison



Why byte-by-byte comparison is no good...

- i. All offsets and branch displacements have to be masked out of the comparison
 - ii. Compilers like to re-arrange basic blocks
 - iii. Objects might be linked in different order
 - iv. Individual instructions might have been replaced by functional equivalents
- All these things will make byte-by-byte comparison report big changes when none really occurred

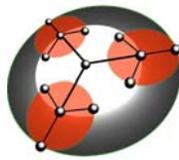
Structural Comparison



Viewing a program as graph of graphs

- i. Primarily one is interested in changes to program logic
- ii. A program can be viewed by looking at two graphs:
 - i. The callgraph which contains all functions and their relationships (A calls B etc.)
 - ii. The individual function flowgraphs which represent the basic blocks of every function and how they are linked by conditional or unconditional branches
- iii. The program logic is more or less encoded in these two graphs
 - i. Adding a single `if()` in any function will trigger a change in it's flowgraph
 - ii. Changing a call to `strcpy` to a call to `strncpy` will change the callgraph

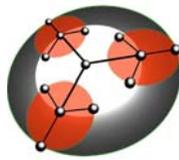
Structural Comparison



Detecting changes by comparing graphs

- i. Program logic can be viewed as a callgraph with nodes representing the individual flowgraphs
- ii. Comparing two executable based on these graphs will detect all logic changes
- iii. The comparison should be false-positive-free:
 - i. Only “real” changes to program logic should be detected
 - ii. Compilers don’t usually change the program logic
 - iii. Modern compilers can inline entire functions and do many more crazy things (thus the “should be” instead of “is”)
- iv. The comparison will not be false-negative-free:
 - i. Switching signedness of a type or changing constants and buffer sizes will go undetected
- v. So how can two graphs of graphs be compared ?

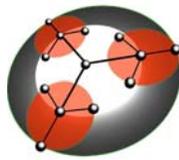
Structural Comparison



Comparing the graphs

- i. Checking if two undirected graphs are isomorphic is NP -- Math-speak for: Finding out if two graphs are the same is damn expensive
- ii. Problem is a lot less problematic if one can find “fixed points” – two nodes in the two graphs that are definitely the same
- iii. When analyzing programs, entry points and names for imported functions are available, yielding a first set of “fixed points”
- iv. More “fixed points” would be desirable – so what would be a decent heuristic to generate more of them ?

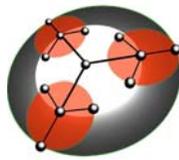
Structural Comparison



Heuristic signatures for the functions

Simplistic signature for every function:

- Number of basic blocks
- Number of links
- Number of functions called from this function



Structural Comparison

Heuristic signatures for the functions: Basic Blocks

5 Nodes

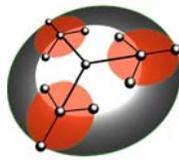
```
75851490:  
push    ebx  
mov     ebx, ds:75851378InterlockedIncrement  
push    esi  
push    edi  
mov     esi, ecx  
lea    edi, [esi+1Ch]  
push    edi ; 1pAddend  
call   ebx ; 75851378InterlockedIncrement  
cmp     dword ptr [esi+20h], 0  
jnz    7586A2BC1ac_7586A2BC
```

```
7586A2BC:  
push    edi  
call   ds:7585137CInterlockedDecrement  
test   eax, eax  
jnz    short 7586A2D01ac_7586A2D0
```

```
7586A2C7:  
push    dword ptr [esi+18h]  
call   ds:7585139CSetEvent
```

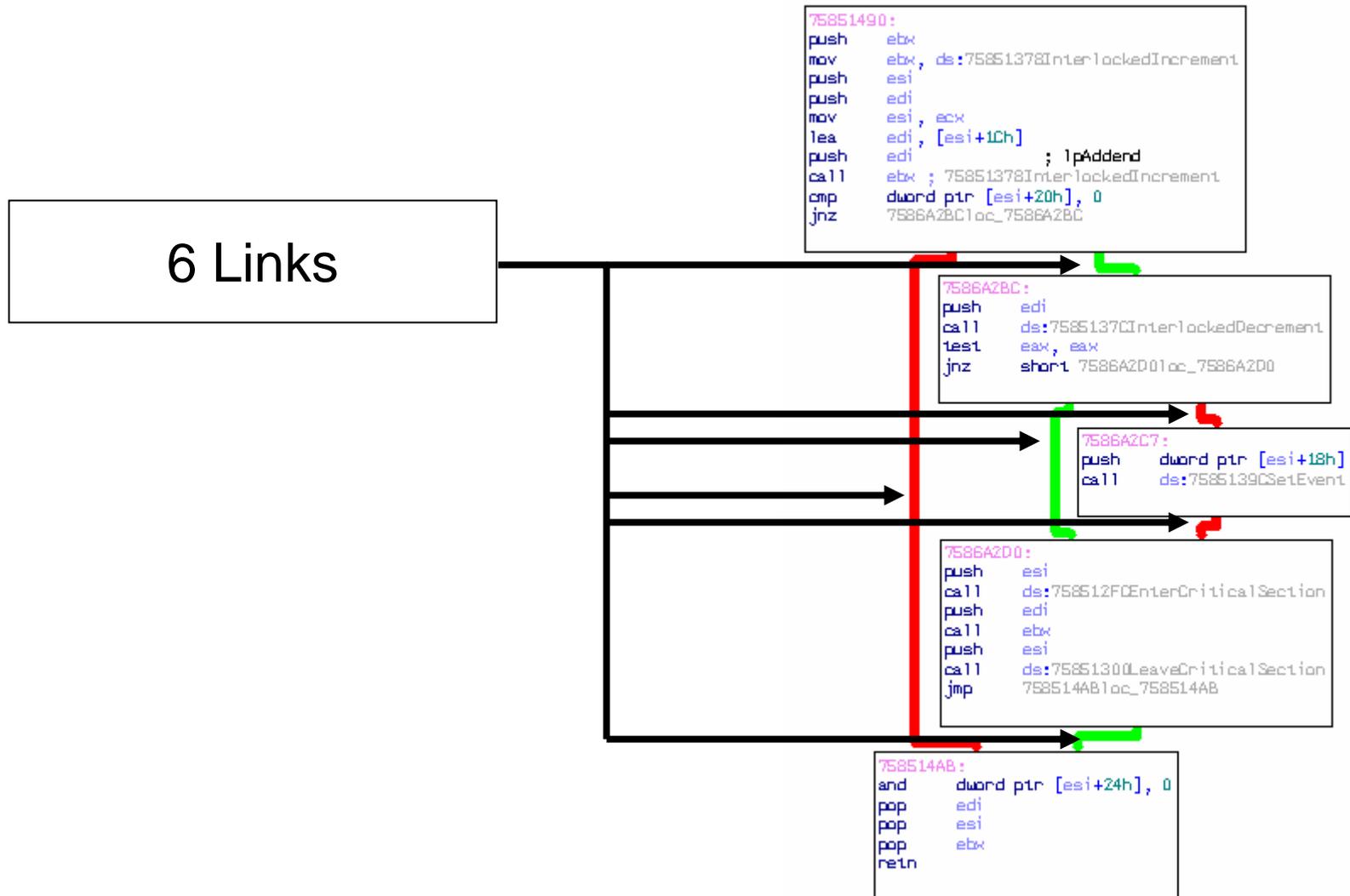
```
7586A2D0:  
push    esi  
call   ds:758512FCEnterCriticalSection  
push    edi  
call   ebx  
push    esi  
call   ds:75851300LeaveCriticalSection  
jmp    758514AB1ac_758514AB
```

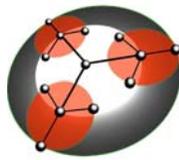
```
758514AB:  
and     dword ptr [esi+24h], 0  
pop     edi  
pop     esi  
pop     ebx  
retn
```



Structural Comparison

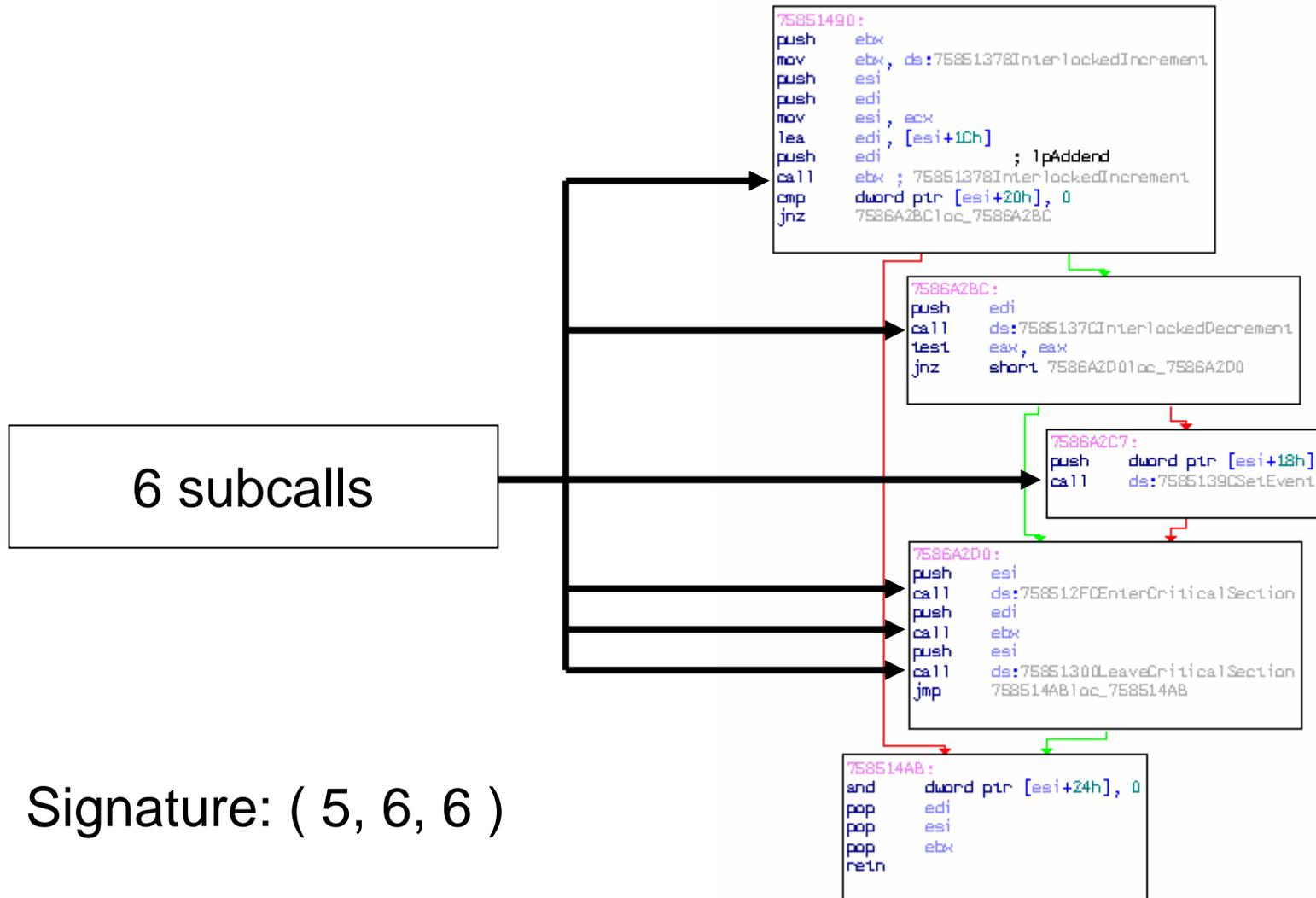
Heuristic signatures for the functions: Links

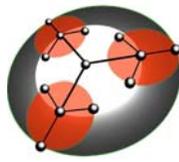




Structural Comparison

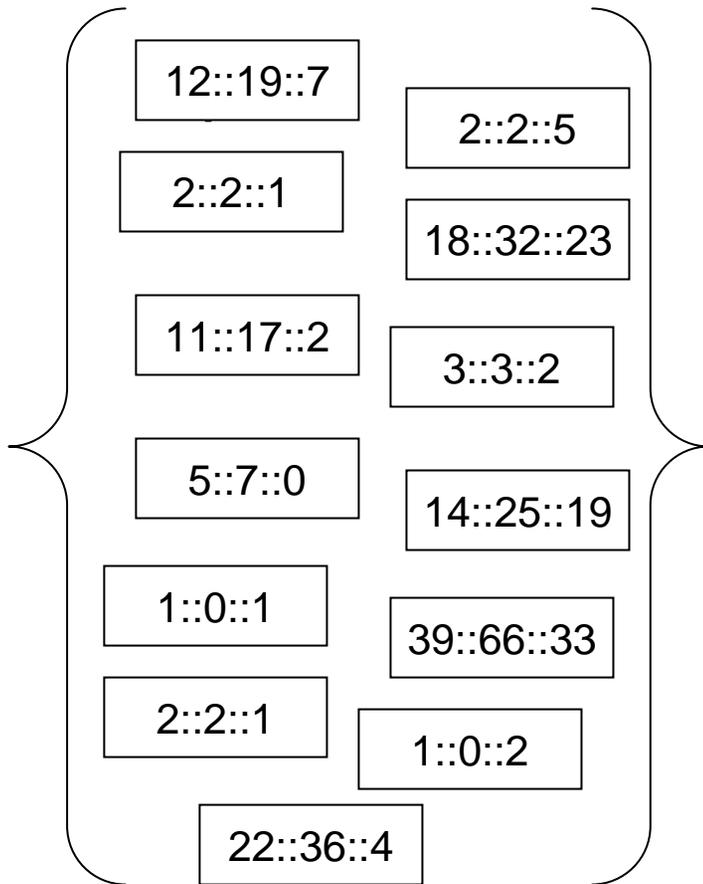
Heuristic signatures for the functions: Subcalls



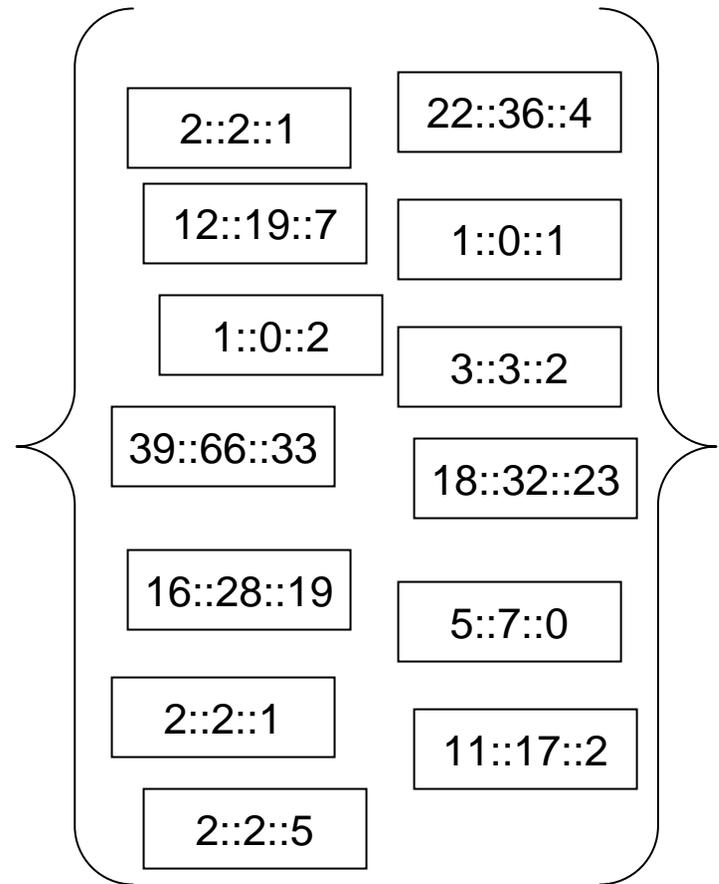


Structural Comparison

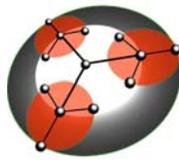
Finding more fixedpoints (1)



Signature Set of Binary A

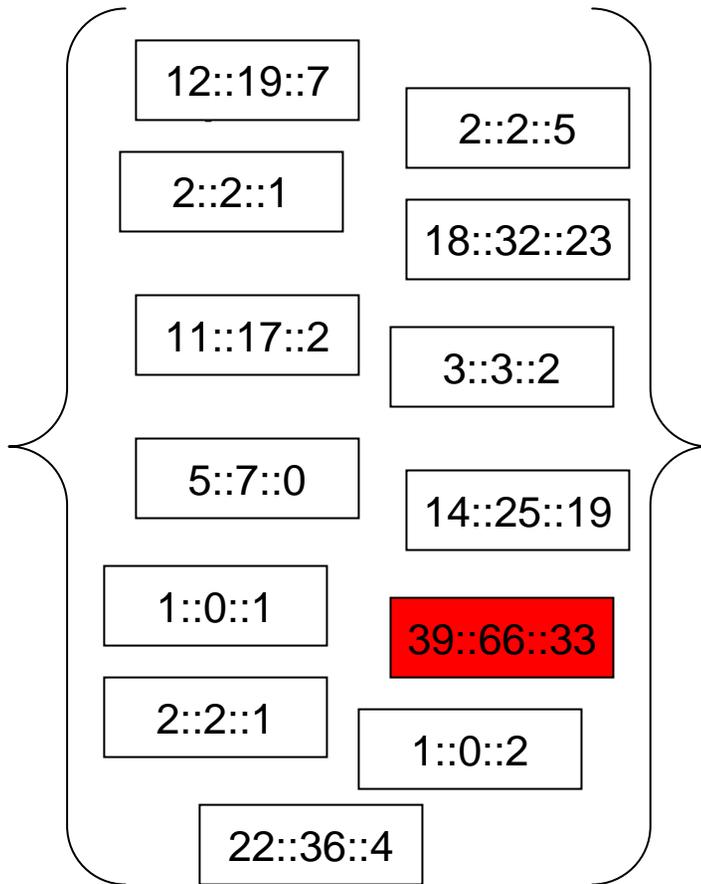


Signature Set of Binary B

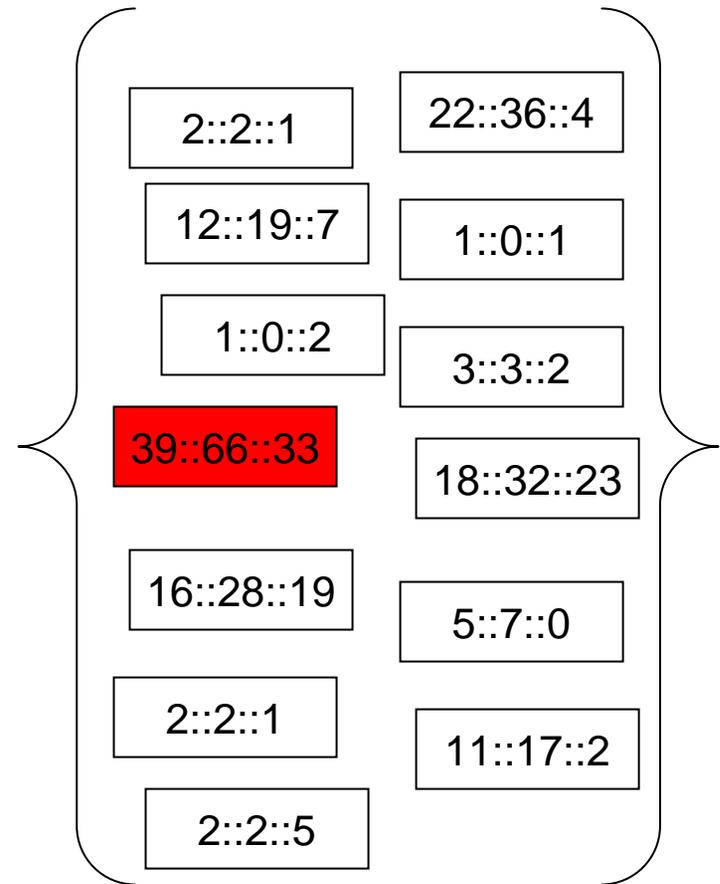


Structural Comparison

Finding more fixedpoints (2)

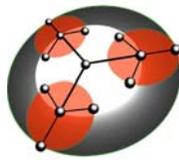


Signature Set of Binary A

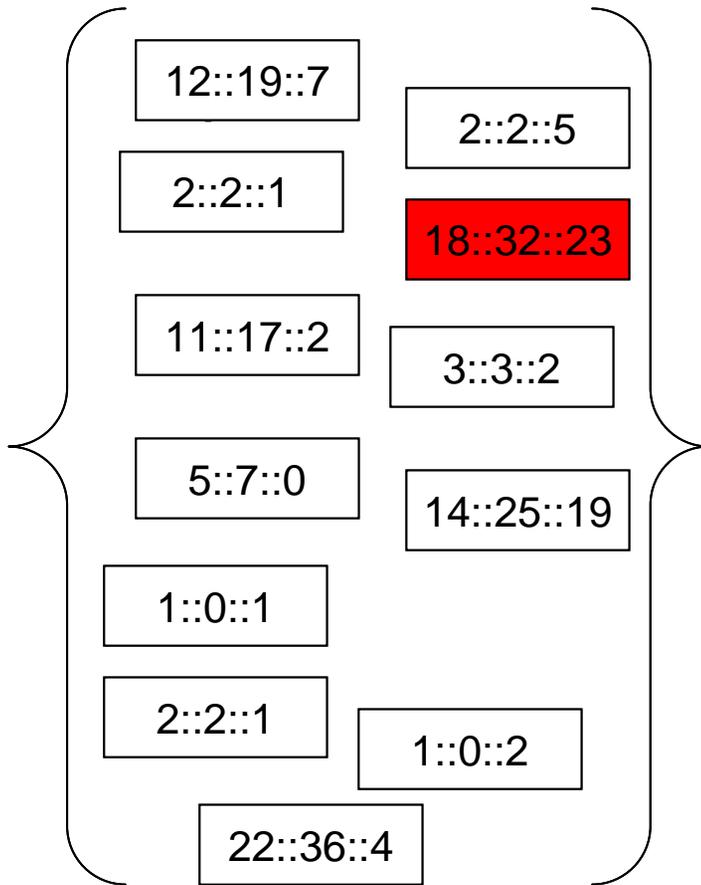


Signature Set of Binary B

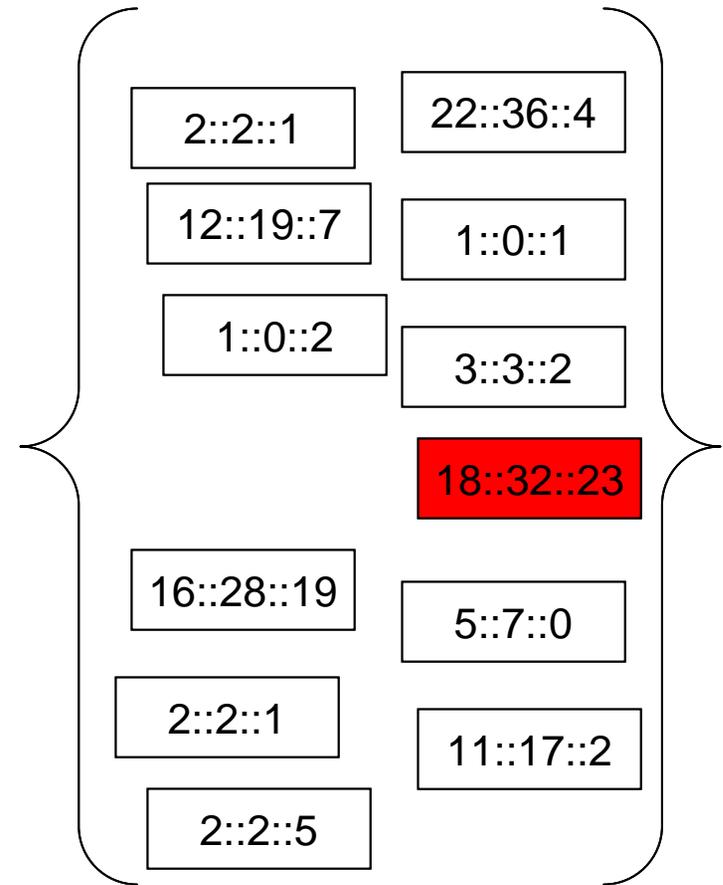
Structural Comparison



Finding more fixedpoints (3)

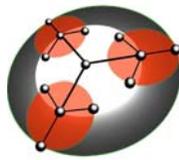


Signature Set of Binary A

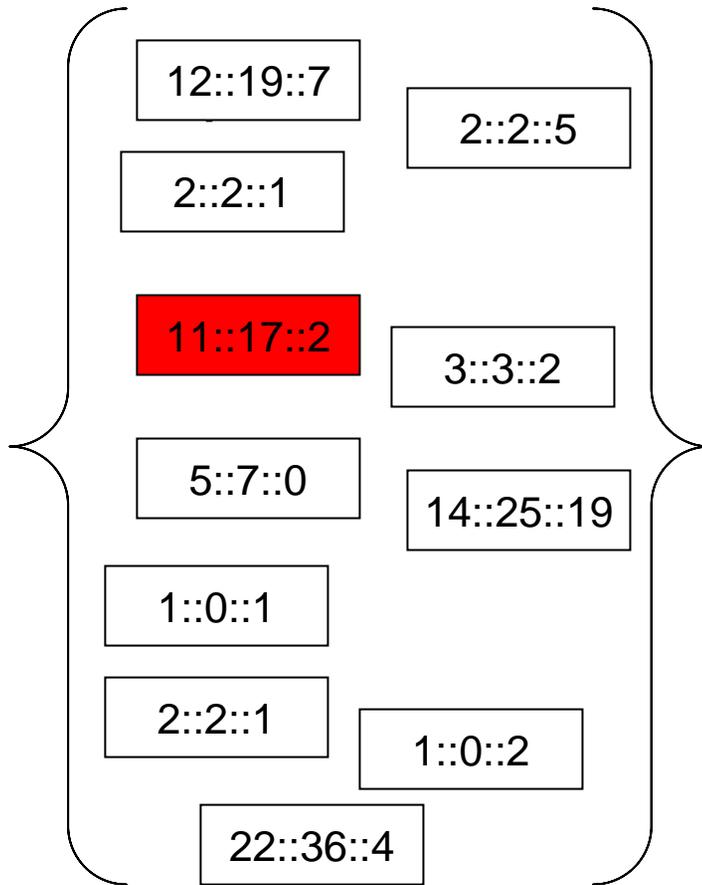


Signature Set of Binary B

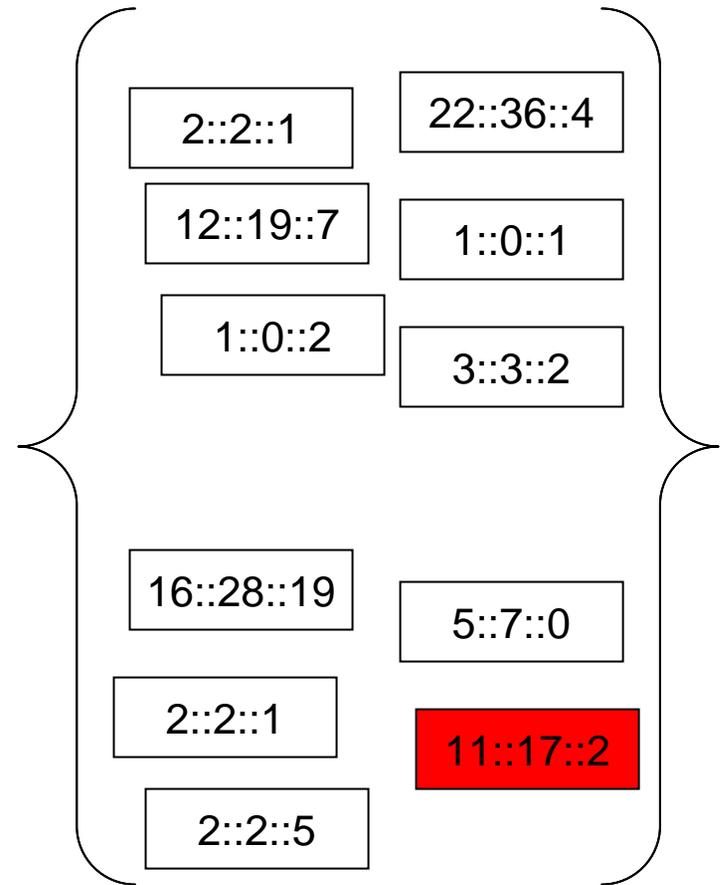
Structural Comparison



Finding more fixedpoints (4)

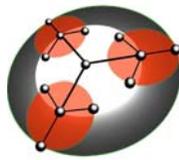


Signature Set of Binary A

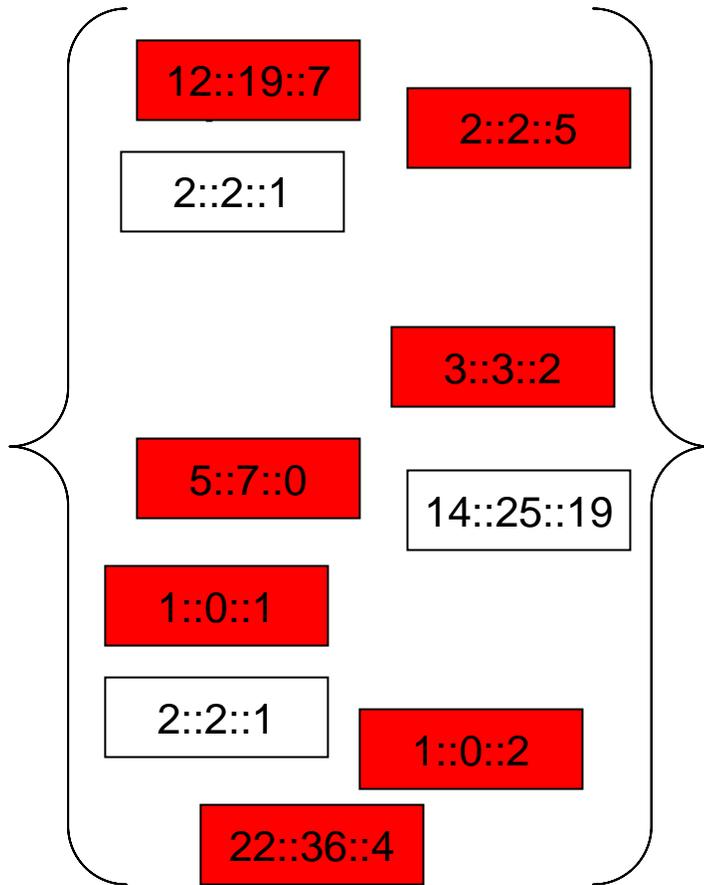


Signature Set of Binary B

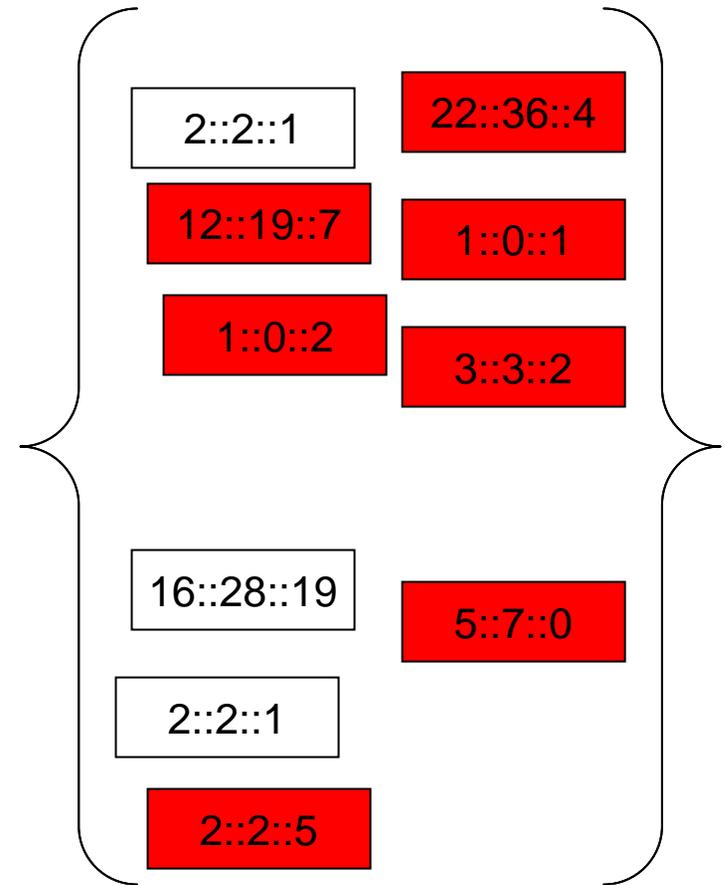
Structural Comparison



Finding more fixedpoints (5)

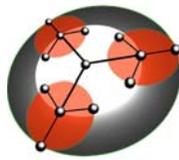


Signature Set of Binary A

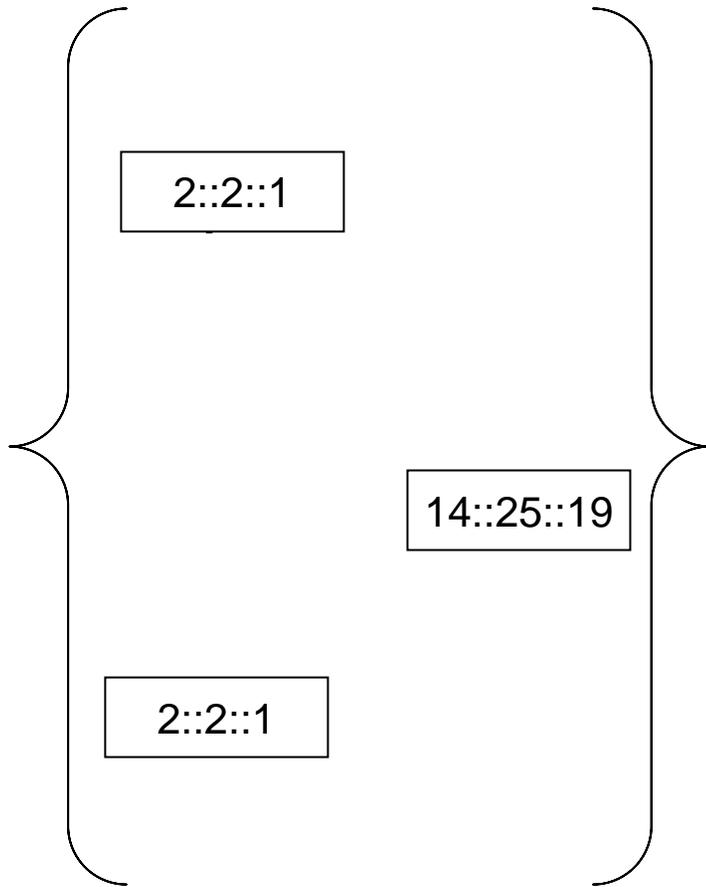


Signature Set of Binary B

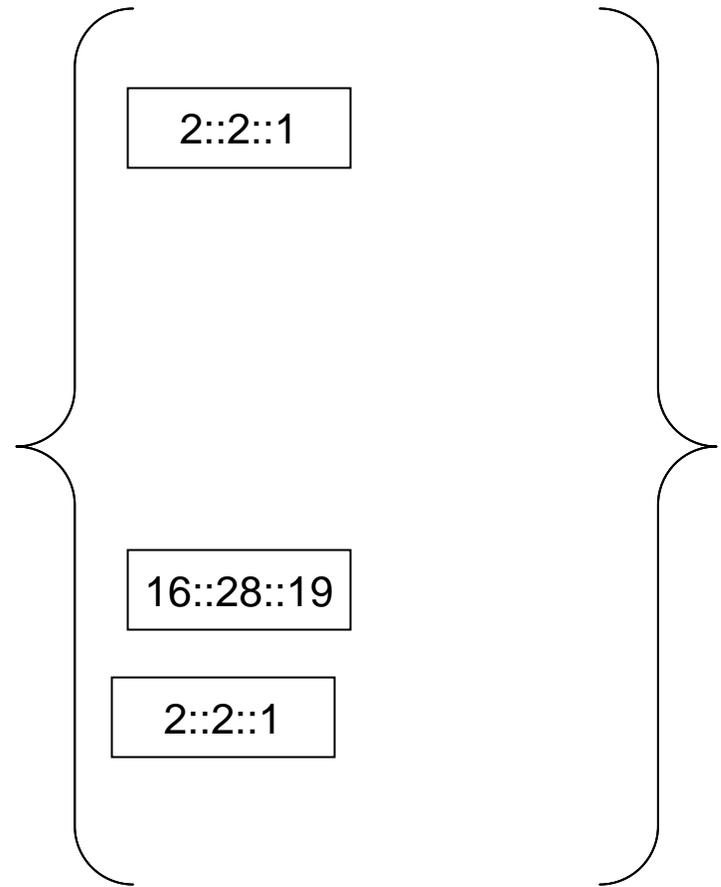
Structural Comparison



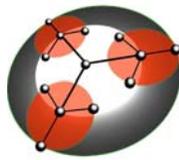
Finding more fixedpoints (6)



Signature Set of Binary A



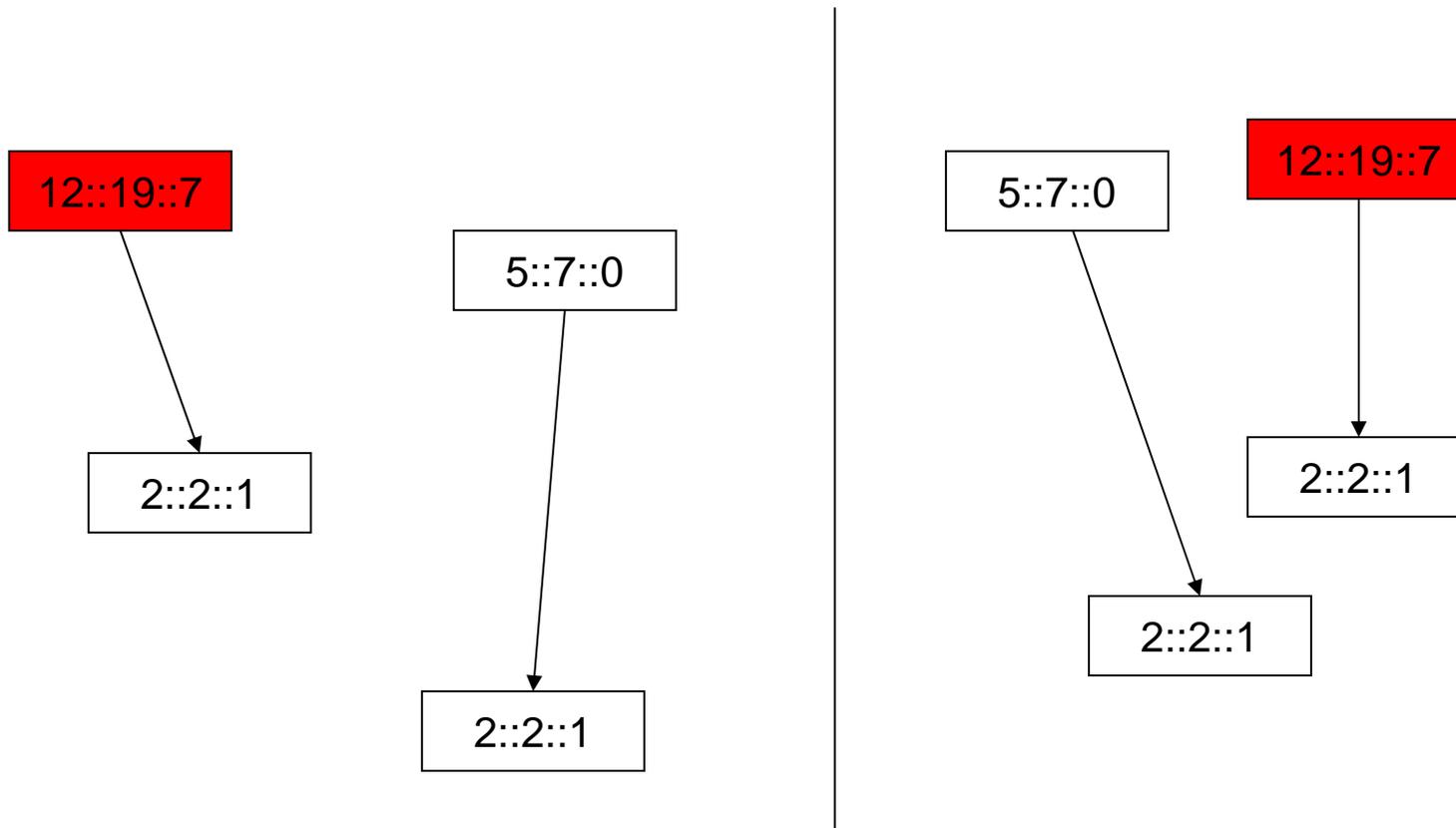
Signature Set of Binary B

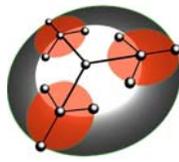


Structural Comparison

Heuristic signatures for the functions: More fixedpoints

In the next step, signatures that occur more than once in both binaries are eliminated by using the callgraph:

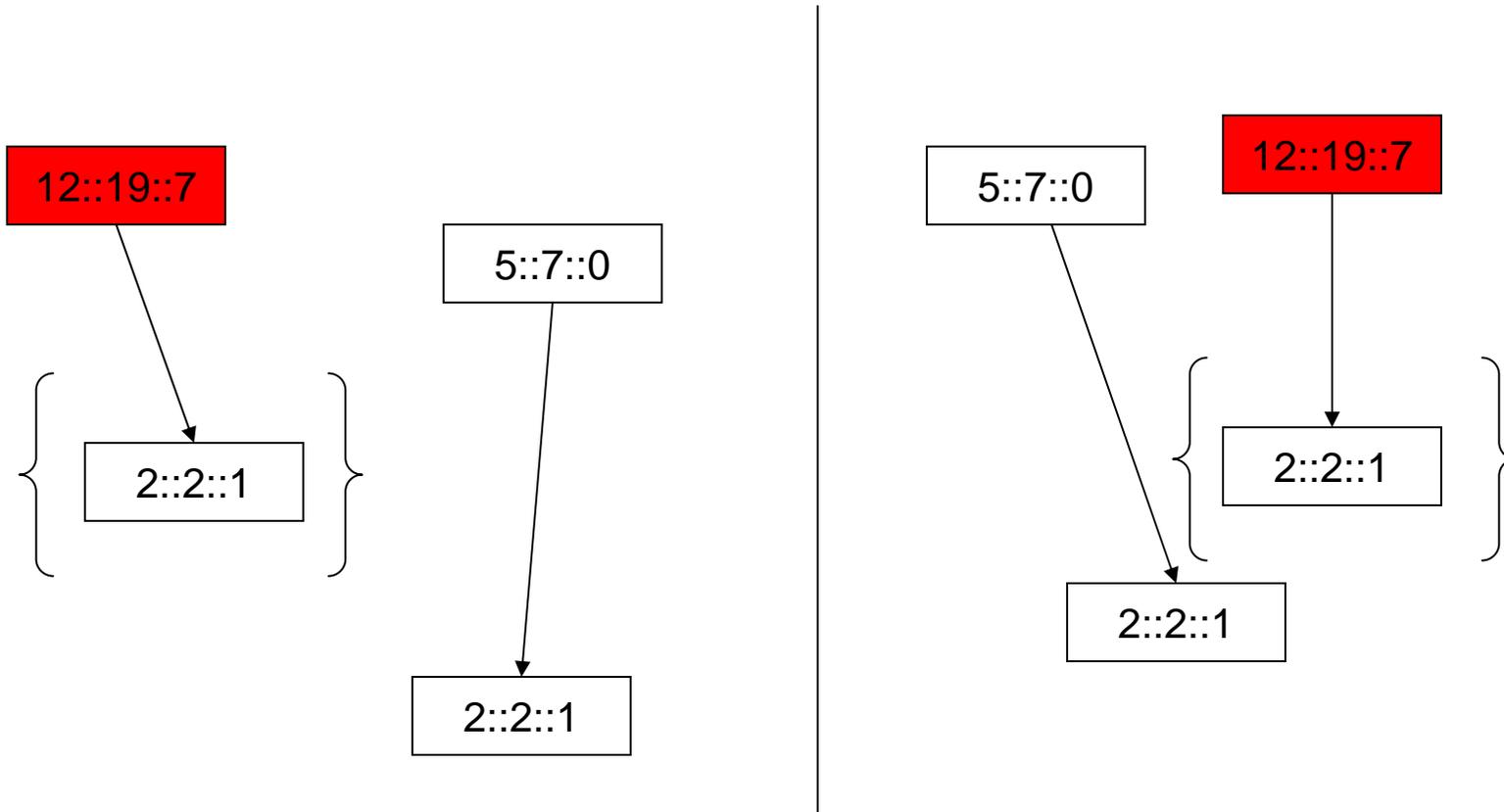




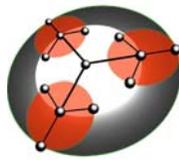
Structural Comparison

Heuristic signatures for the functions: More fixedpoints

In the next step, signatures that occur more than once in both binaries are eliminated by using the callgraph:



Structural Comparison

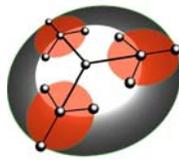


More Fixedpoints

It's clear that smaller subsets of functions will have fewer “collisions”, and thus generate more fixedpoints. One can restrict the sizes of the examined sets using many different characteristics:

- Same “indegree” in the callgraph
- Same “outdegree” in the callgraph
- Reference the same string in both binaries
- (...)

Structural Comparison



Small Prime Products

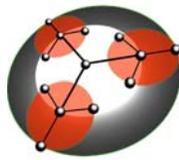
Compilers tend to re-order instructions for better pipelining and optimization a lot

- A quick algorithm to see if two sequences of instructions contain the same instructions, just re-ordered, would be nice
- A “signature” that contains the instructions but not their order and is reasonably short/easy to represent

Small Prime Products (SPP):

- Each instruction type (mov, lea etc.) is assigned a small prime number
- For a given sequence, all the primes corresponding to the instructions in the sequence are multiplied together
- Because of uniqueness of prime number decompositions and because $a*b = b*a$ we have all possible permutations covered

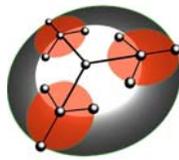
Structural Comparison



A fun example: ISA Server H323ASN1.DLL

- i. A new check on decoded integers before entering a loop *yawn*
 - ii. Additional checks to check the input of a function called
ASN1PERDecZeroCharTableStringNoAlloc()
 - iii. Further inspection yields integer overflow in that function
 - iv. ... hrm ... shouldn't one fix the library instead of the application ? What other applications might use this
- ➔ Free NetMeeting remote bug which was fixed only a few months later...

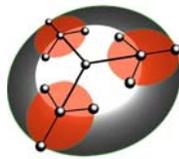
Structural Comparison



Another fun example: SCHANNEL.DLL

- i. No information except “bug in PCT parsing”
- ii. Running BinDiff yielded a bunch of results, but only one that had “PCT” in the function name
- iii. Funny bug: Overwrite EIP with anything, including NULL bytes...
- iv. Total time invested to locate and understand the bug after receiving the patch: Less than one hour
- v. ... more about this later in this talk...
- vi. For those into reading more in-depth papers:

http://www.sabre-security.com/files/dimva_paper2.pdf



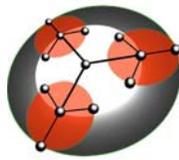
Porting Symbols

Malware analysis...

- More and more malware hits the networks every day
- Malware authors have adapted to a “multi-version”-development cycle
- The source code of “Agobot” etc. was widely distributed, and many special variants exist

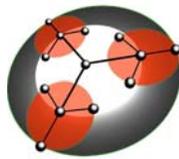
The presented algorithms can be used to:

- Re-use information from a disassembly of an earlier variant of a piece of malware
- Measure similarity between code bases, and thus uncover cooperation between malware authors
- In the future: Automatically classify new malware into the correct “malware family”, by measuring code similarity & clustering



Porting Symbols

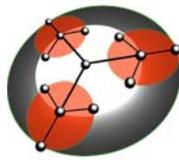
Malware analysis (Example)



Porting Symbols

Porting library symbols into ROMs

- When analyzing embedded systems (or in fact any sort of binary), a lot of code one encounters is clearly from open sources
- Things like OpenSSL are VERY common in many applications
- It would be nice if we could make use of that information
- We can compile OpenSSL with debug symbols, and “back-port” the debug symbols into our disassembly, even though the compiler was different



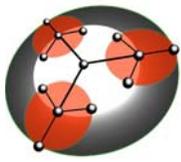
Porting Symbols

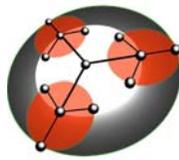
Porting library symbols into ROMs

Example

Navigating Binaries

Road Maps as analogy for programs

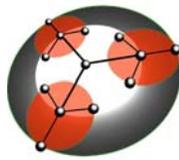




Navigating Binaries

Road Maps as analogy for programs

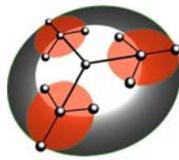
- i. When driving in a car, a frequent problem is to get from the current location of the car to another location
- ii. Similar problems have to be solved in security analysis:
 - i. A patch for a problem changes a function somewhere deep in program logic and fixes a security problem
 - ii. A (static) analysis tool detects a problem somewhere deep in program logic
- iii. In both cases, the problem of finding a way from point A to point B has to be solved
 - i. Why use textual representations such as “road A leads to road B” etc. for program analysis ? There is a good reason we invented maps...



Navigating Binaries

The importance of visualisation

- i. Programs are huge and not easily understood
- ii. Most of the human brain is built to react to visual stimulus
 - i. Humans are more suited to recognize food than to keep large graphs in their head
 - ii. “Recognition” tasks are a lot faster than “memory” tasks, meaning that reading code dependencies is significantly slower than “seeing” them
- iii. Many problems are data visualisation problems
- iv. Good ways to visualise function dependencies can yield better understanding

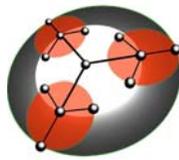


Navigating Binaries

The differences between road maps and programs

- i. Roads crossings have an outdegree of less than 4 usually
 - i. Many functions call much more than just 4 subfunctions
- ii. Road crossings have in indegree of less than 4 usually
 - i. Many functions get called from a lot more than 4 subfunctions
- iii. When driving with a car, one usually does not drive down many dead ends
 - i. A subfunction call that does not lead to the desired location immediately will nonetheless be executed

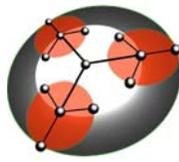
➔ Useful analogy, but use with care



Navigating Binaries

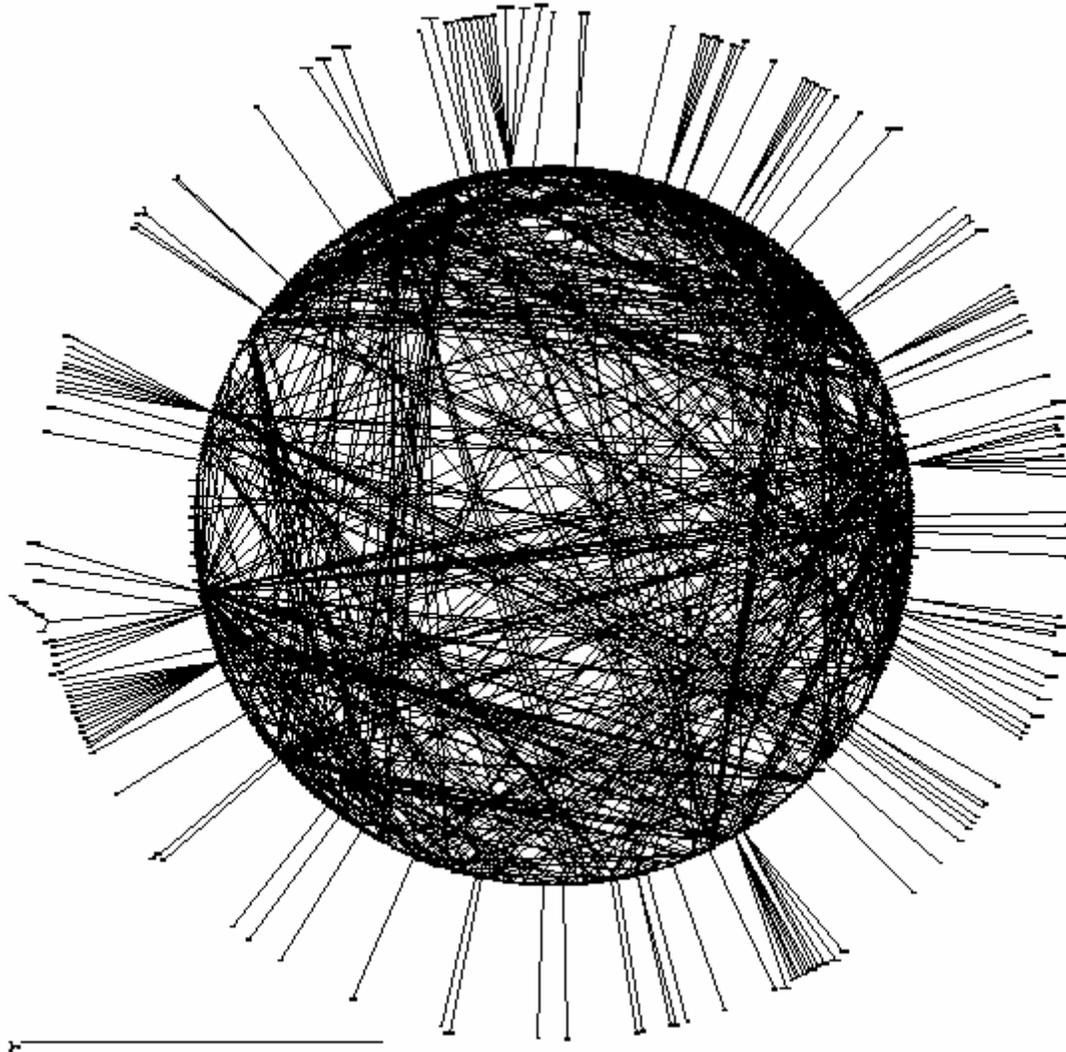
Restructuring callgraphs

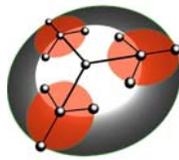
- i. Library functions (such as `malloc()`) tie together logically independent parts of the binary in the graph
 - ii. Removal of library functions should clean up the graph significantly
 - i. Library functions are called from many locations, thus adding a significant number of edges
 - ii. Library functions have callee's grouped closer together even though no obvious logical connection exists
 - iii. Removing via blacklists is a bad idea
 - i. Not generic enough
 - ii. "Wrappers" will remain
- ➔ Removal of nodes dependent on in/outdegree



Navigating Binaries

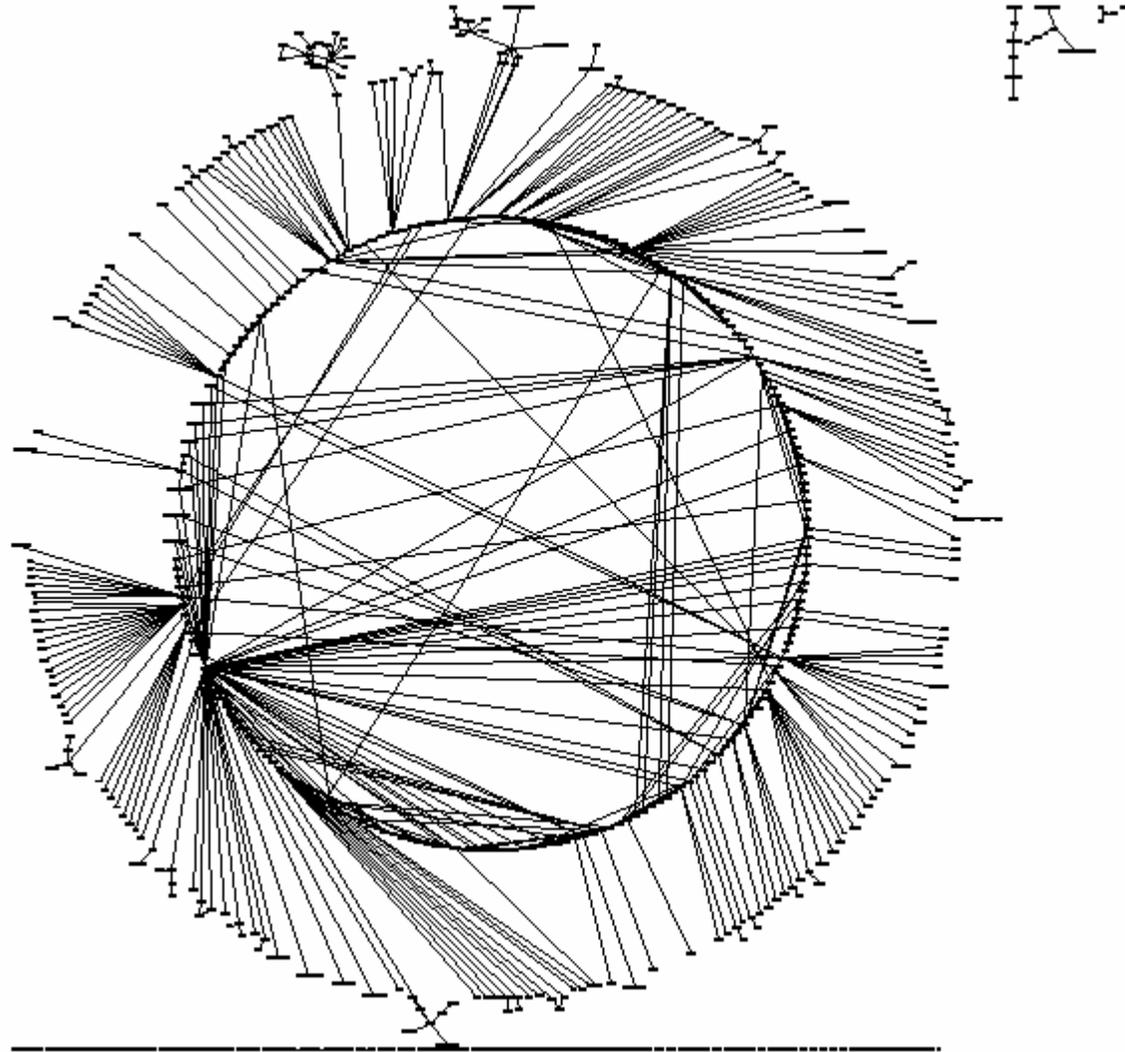
Restructuring callgraphs

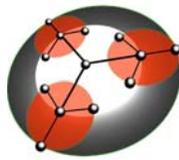




Navigating Binaries

Restructuring callgraphs



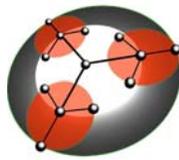


Navigating Binaries

Getting your bearings in an unknown binary

Auditing binaries is like harpooning in cold, dark water:

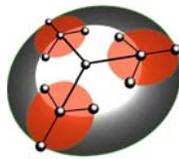
- You have a hard time telling what your surroundings look like
 - You have a hard time telling where exactly you are
 - A map is only helpful if you can somehow find out where on the map you yourself are
-
- ➔ A callgraph is a “map” of the executable
 - ➔ We can do find out where we are by setting “echo-breakpoints” – breakpoints in a small debugger that, when hit, send the address that was hit over the wire



Navigating Binaries

Navigating using “echo”

- i. Same concept as presented at BH Vegas 2002
 - ii. Set BPX on all nodes (remove upon hit)
 - iii. Visualize the results in a graph
 - i. Highlight all functions that have been hit
 - ii. Remove nodes that cannot be reached from the nodes that have been hit
 - iii. Playback the path the program takes as a “movie”
 - iv. Attempt to navigate to a certain location using your map and your own location
- ➔ Navigation can be significantly improved



Navigating Binaries

Other uses (Library Functions II)

- i. We can identify library functions just based on their properties in the graph (indegree/outdegree)
- ii. Library functions are of special interest when auditing code in order to find vulnerabilities:
 - i. Library functions are called from many locations – if they contain a bug, it is likely that a function with high indegree is reachable from a location that we can hit
 - ii. Library functions are called from many locations – if they react badly to certain inputs, the sheer number of calls increase the odds that one can get malicious input in

End of the talk

Any questions ?

