

How to write a basic control flow decompiler

By Warranty Voider

1. Abstract

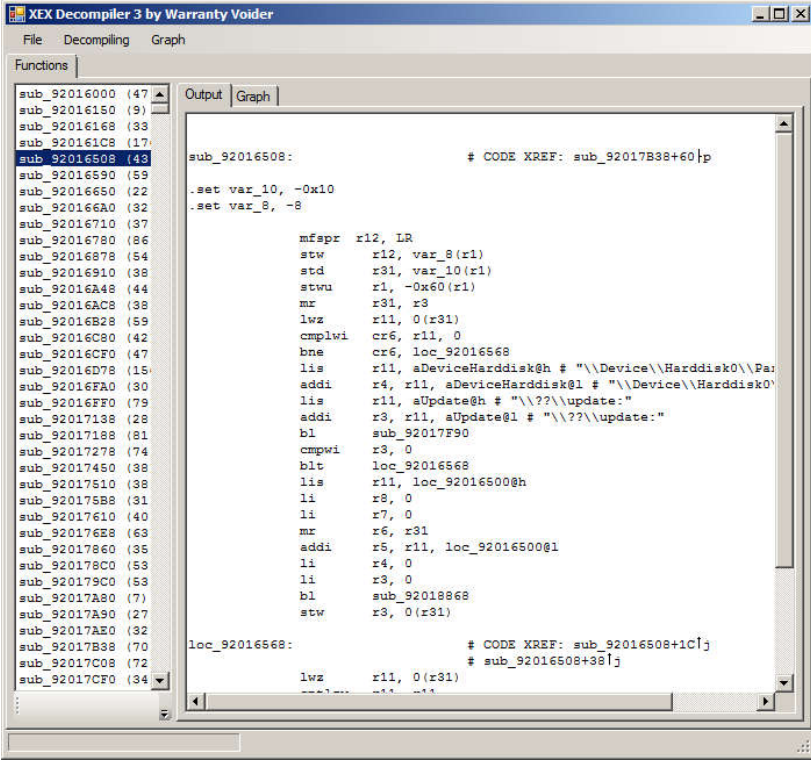
In this paper it is shown, how to write a general decompiler for the control flow of a given sub function. It assumes as input an assembler code file from a disassembler that already converted binary opcodes to mnemonic representation and added location labels for jumps inside a sub function. This can be made part of the decompiler and put as step before the start, but isn't focus of this paper.

In the following section it is explained how decompiling of the control flow can be achieved from such a asm listing, by converting it from a listing problem to a graph problem and from there convert the graph problem to a tree problem, at which point it can be "decompiled" out back to a readable language.

This paper does not explain how dataflow reversing can be achieved, as this can be added as an extra step before or after the control flow recovery, instead the asm opcodes are grouped into blocks of assembler instructions that are guaranteed to be executed at once and not jumped into. These blocks can be further processed into abstract representation of its internal dataflow, but for this paper they are simply written out as sub functions.

This paper is based on the paper "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations" (ref #1) and my experience on applying it on the PPC architecture of Xbox 360 XEX files (ref #2). This paper does not cover the transformations for multi-exit and -entry regions, but they could be added with further processing. It is therefore not always possible to decompile a sub function completely, but if it does finish, it is guaranteed to be "goto-free".

2. The Listing Problem – getting the Control Flow Graph



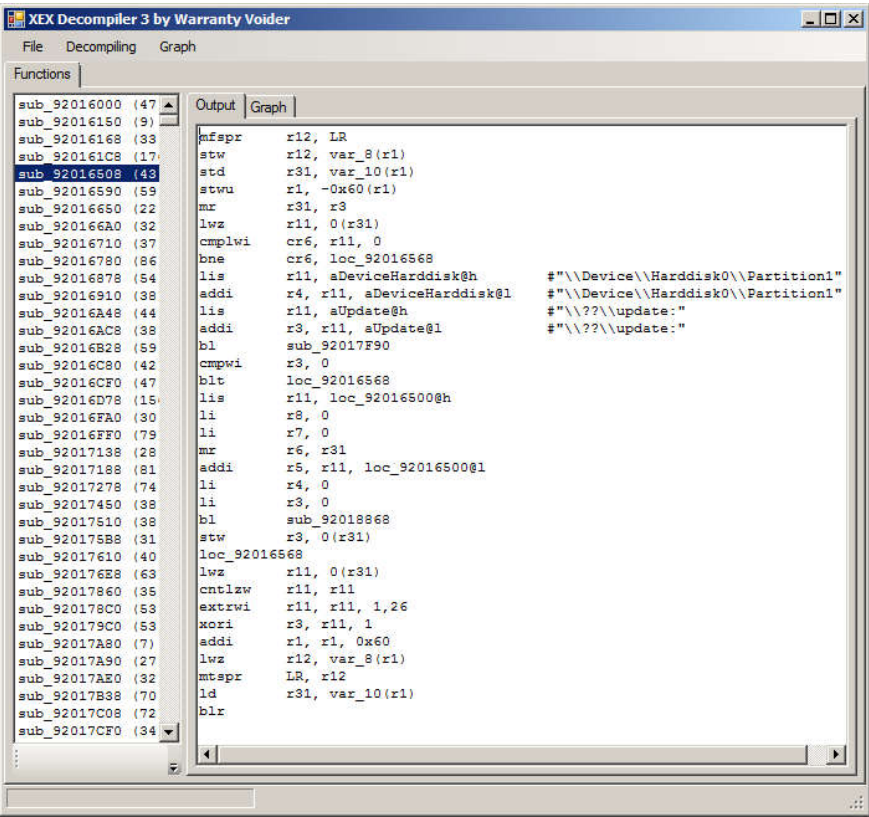
The screenshot shows the XEX Decompiler 3 by Warranty Voider interface. The 'Functions' list on the left includes sub_92016000 (47), sub_92016150 (9), sub_92016168 (33), sub_920161C8 (17), sub_92016508 (43), sub_92016590 (59), sub_92016650 (22), sub_920166A0 (32), sub_92016710 (37), sub_92016780 (86), sub_92016878 (54), sub_92016910 (38), sub_92016A48 (44), sub_92016AC8 (38), sub_92016B28 (59), sub_92016C80 (42), sub_92016CF0 (47), sub_92016D78 (15), sub_92016FA0 (30), sub_92016FF0 (79), sub_92017138 (28), sub_92017188 (81), sub_92017278 (74), sub_92017450 (38), sub_92017510 (38), sub_920175B8 (31), sub_92017610 (40), sub_920176E8 (63), sub_92017860 (35), sub_920178C0 (53), sub_920179C0 (53), sub_92017A80 (7), sub_92017A90 (27), sub_92017AE0 (32), sub_92017B38 (70), sub_92017C08 (72), and sub_92017CF0 (34). The 'Output' tab is selected, showing assembly code for sub_92016508. The code includes comments like '# CODE XREF: sub_92017B38+60|p' and '# CODE XREF: sub_92016508+1C|j' and '# sub_92016508+38|j'. The code is as follows:

```
sub_92016508:                                # CODE XREF: sub_92017B38+60|p
.set var_10, -0x10
.set var_8, -8

        mfspr    r12, LR
        stw      r12, var_8(r1)
        std      r31, var_10(r1)
        stwu     r1, -0x60(r1)
        mr       r31, r3
        lwz      r11, 0(r31)
        cmplwi   cr6, r11, 0
        bne      cr6, loc_92016568
        lis      r11, aDeviceHarddisk@h # "\\Device\\Harddisk0\\Pa
        addi     r4, r11, aDeviceHarddisk@l # "\\Device\\Harddisk0
        lis      r11, aUpdate@h # "\\??\\update:"
        addi     r3, r11, aUpdate@l # "\\??\\update:"
        bl       sub_92017F90
        cmpwi    r3, 0
        blt      loc_92016568
        lis      r11, loc_92016500@h
        li       r8, 0
        li       r7, 0
        mr       r6, r31
        addi     r5, r11, loc_92016500@l
        li       r4, 0
        li       r3, 0
        bl       sub_92018868
        stw      r3, 0(r31)

loc_92016568:                                # CODE XREF: sub_92016508+1C|j
                                                # sub_92016508+38|j
        lwz      r11, 0(r31)
```

Initially the asm text output of most disassemblers is filled with additionally comments, free lines and annotations that are not needed for the decompiler. Only location labels and lines with mnemonic opcodes are of interest. So as first step the text gets normalized into a format that can be processed automatically.



The screenshot shows the XEX Decompiler 3 by Warranty Voider interface. The 'Functions' list on the left is the same as in the previous screenshot. The 'Output' tab is selected, showing normalized assembly code for sub_92016508. The code is as follows:

```
sub_92016508:                                # CODE XREF: sub_92017B38+60|p
.set var_10, -0x10
.set var_8, -8

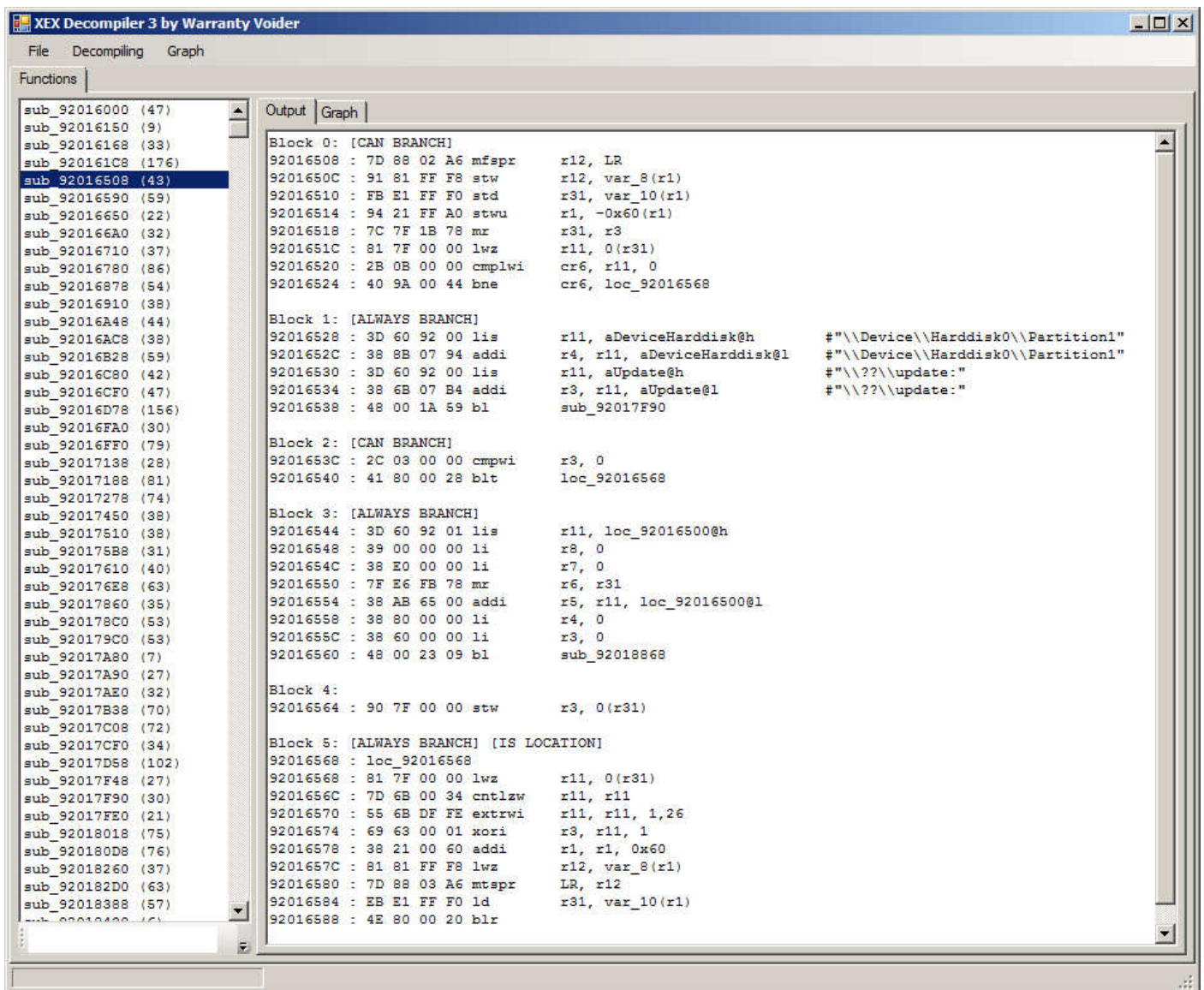
        mfspr    r12, LR
        stw      r12, var_8(r1)
        std      r31, var_10(r1)
        stwu     r1, -0x60(r1)
        mr       r31, r3
        lwz      r11, 0(r31)
        cmplwi   cr6, r11, 0
        bne      cr6, loc_92016568
        lis      r11, aDeviceHarddisk@h # "\\Device\\Harddisk0\\Partition1"
        addi     r4, r11, aDeviceHarddisk@l # "\\Device\\Harddisk0\\Partition1"
        lis      r11, aUpdate@h # "\\??\\update:"
        addi     r3, r11, aUpdate@l # "\\??\\update:"
        bl       sub_92017F90
        cmpwi    r3, 0
        blt      loc_92016568
        lis      r11, loc_92016500@h
        li       r8, 0
        li       r7, 0
        mr       r6, r31
        addi     r5, r11, loc_92016500@l
        li       r4, 0
        li       r3, 0
        bl       sub_92018868
        stw      r3, 0(r31)

loc_92016568:                                # CODE XREF: sub_92016508+1C|j
                                                # sub_92016508+38|j
        lwz      r11, 0(r31)
        cntlwz   r11, r11
        extrwi   r11, r11, 1, 26
        xori     r3, r11, 1
        addi     r1, r1, 0x60
        lwz      r12, var_8(r1)
        mtspr    LR, r12
        ld       r31, var_10(r1)
        blr
```

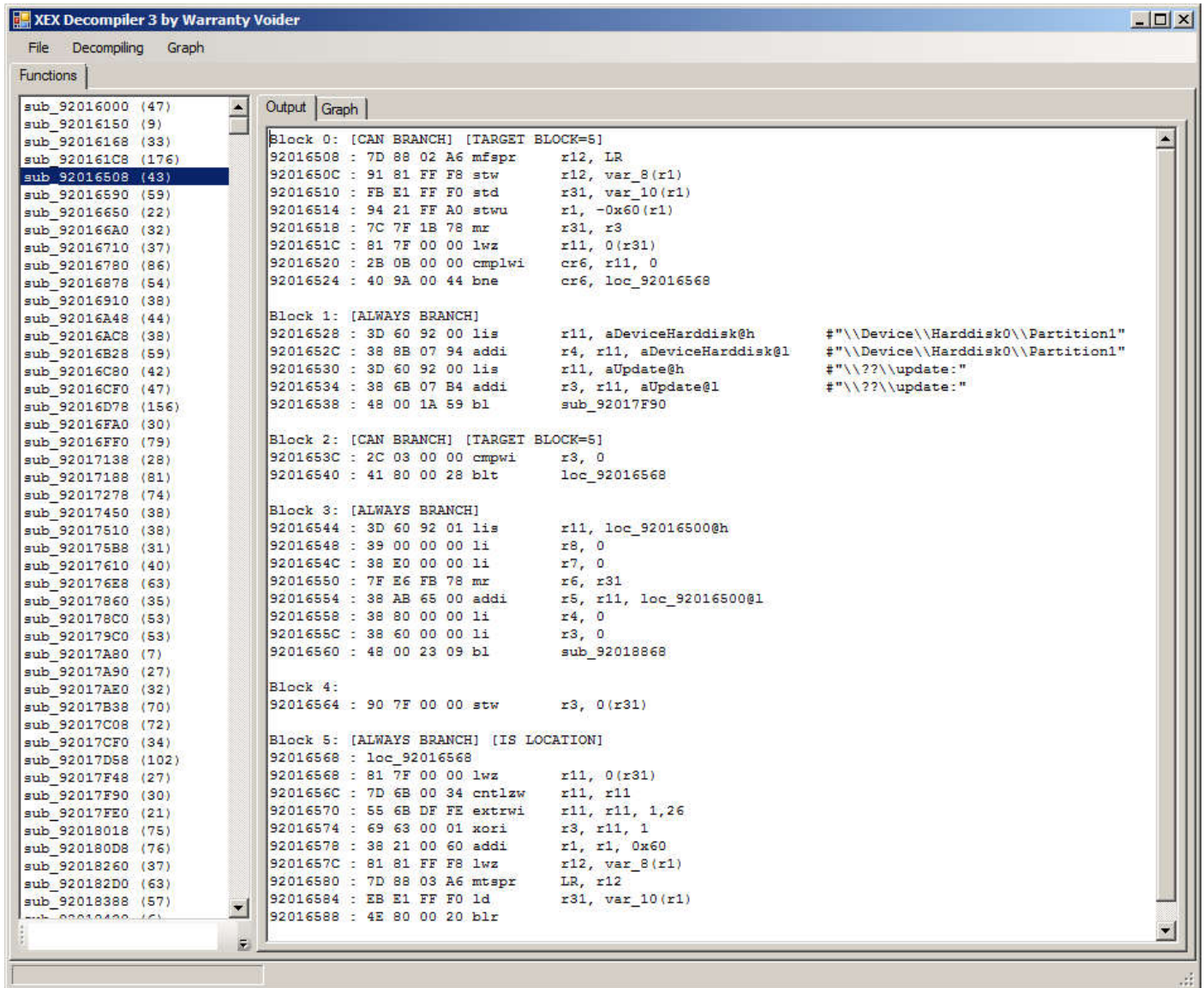
Now this normalized input is a listing of instruction and jump labels, so in order to be able to convert this into a directed graph, it is necessary to group instructions into blocks that get executed at once and that are not jumped into from outside. This guarantees that the control flow can only change at the end of the block, which then can be presented as an directed edge to another block.

In order to group the blocks the input is scanned line wise for branch instruction and jump labels. Once such a location is found, it is determined what kind of block the instructions so far where and they get grouped and annotated by that.

- If the location is a label or is a unconditional branch instruction, the block is considered a simple unconditional block
- If the location is a conditional branch instruction, the block is considered a conditional block
- If the location is a label, the label is part of the next block, otherwise the location is added to the current block and a new block gets started

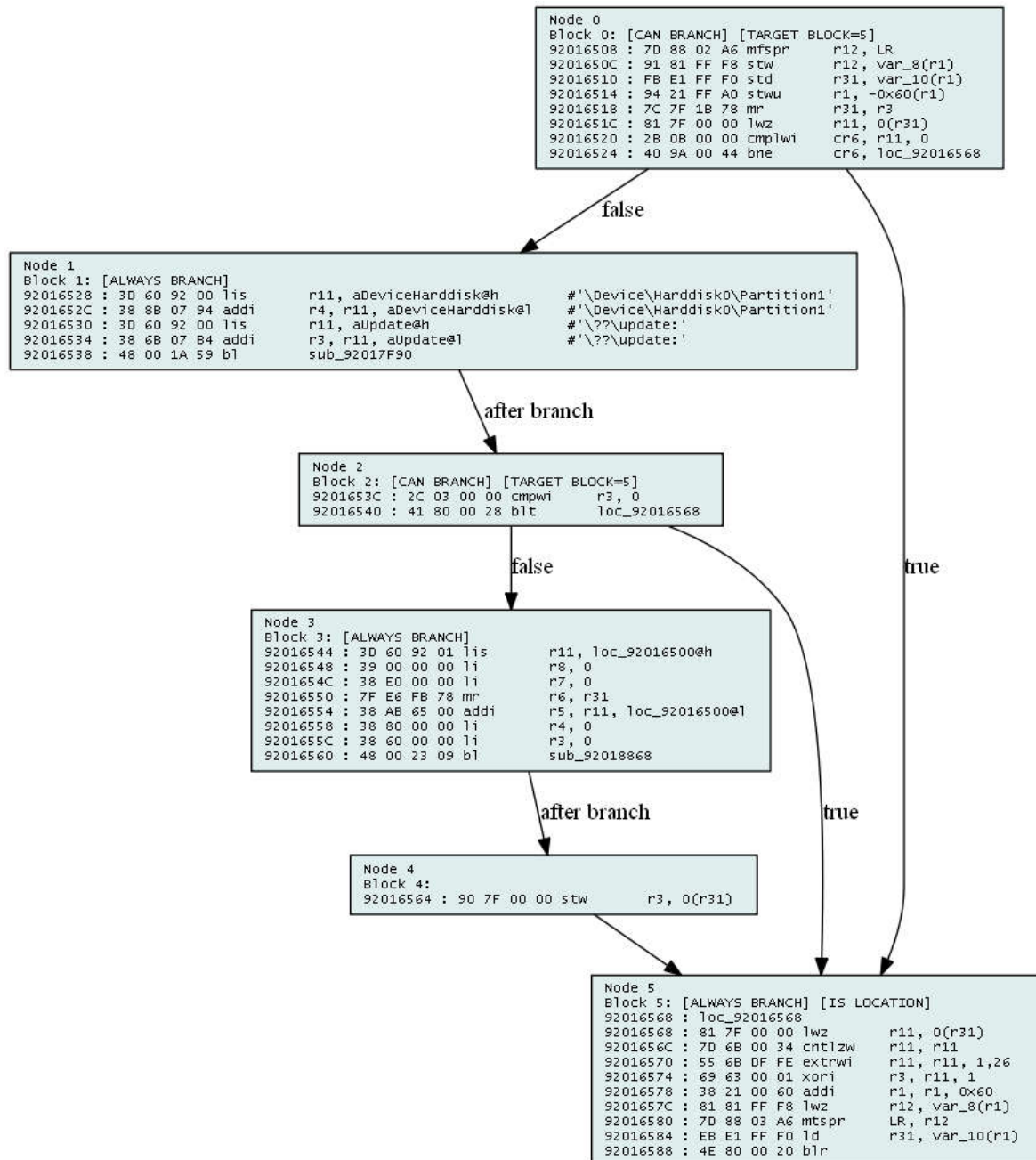


These blocks, called “codeblocks” in the rest of the paper, will be the base for the nodes in the control flow graph. To find the edges for the graph, the change in control flow from block to block is analyzed and for branches inside the sub function the target block number is found.



- If a block [CAN BRANCH] to another block, one edge to that block with the label “true” and one to the immediate following block with label “false” are added
- If a block does [ALWAYS BRANCH], it is checked if the target is inside the sub function. If it is inside, an edge to that target block is added, otherwise its assumed to be a call to another sub function that returns, so an edge to the immediate next block is added
- If none of the above applies an simple edge to the next block (if existent) is added

3.The Graph Problem – getting the AST from the graph



With this the control flow inside the sub function can already be rendered into a graph and gives a better overview. The graph is simply composed of a list of edges and a list of nodes. Each node is a placeholder for an object and now each node has simply a codeblock attached. The idea is to remove groups of nodes and replace them with an abstract node that holds a representation of their inner flow.

This is done until there is only one node left that then contains either a single abstract block or just a codeblock alone. In case of an abstract block, it contains the abstract syntax tree of the sub function. This then just has to be printed out in a human readable language, like a pseudo-C variant.

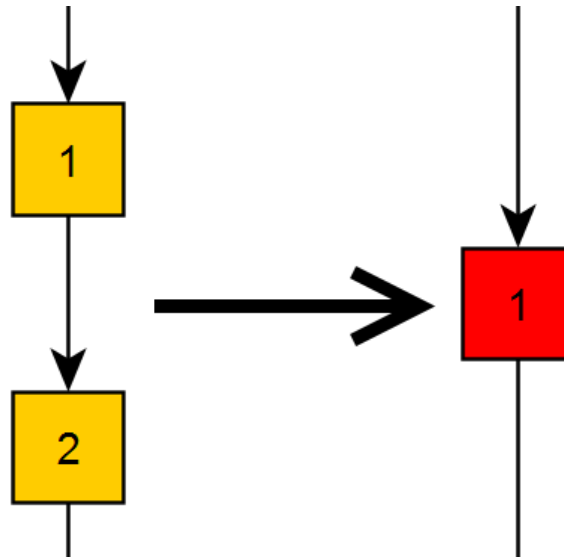
To guarantee that any block that gets reduced has no descendants that haven't been processed yet, the graph is traveled node wise in post order, in the example graph this would be simply:

Post Order = {4, 3, 2, 1, 0}

To get this list a depth first search is done starting from the end of the graph. For this the graph is inverted. (Ref #3)

Then for each node in the list one of the following reduction is tried to be applied

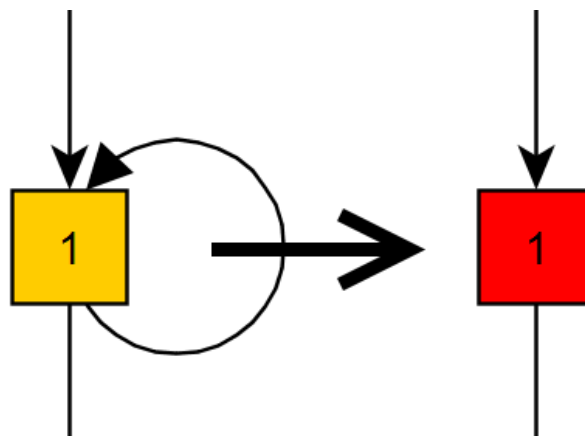
- **Reducing sequential blocks**



One of the easiest reductions is the reduction of sequential blocks, for this following has to be true:

- Block1 has only one child
- Block2 has less than 2 children
- Block2's child is not Block1

- **Reducing "1-Block while" constructs**

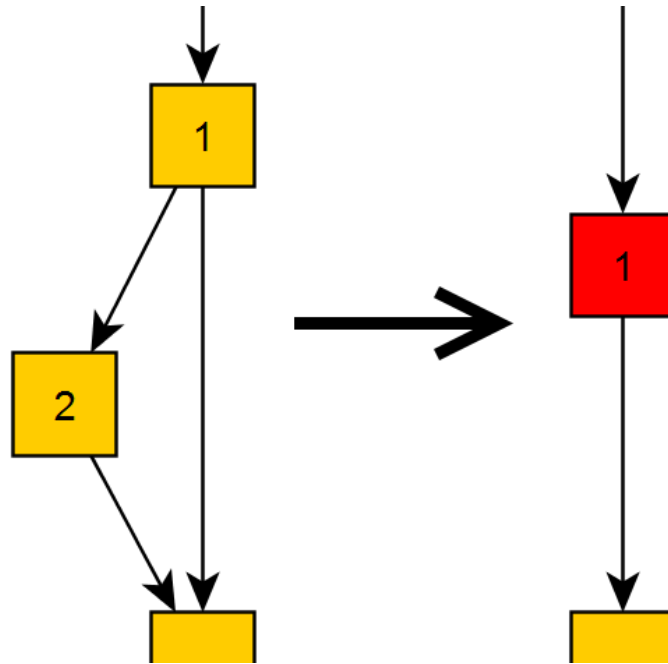


Another easy replacement is the reduction of simple one-block whiles. This helps reducing the number of children, so it can later be picked up by a sequence reduction for example. Notice that the edge isn't simply removed, the codeblock is replaced with an abstract block, saying it's a simple one-block while.

Requirements:

- Block has itself as a parent
- Block has itself as a children

- Reducing “If-Then” constructs

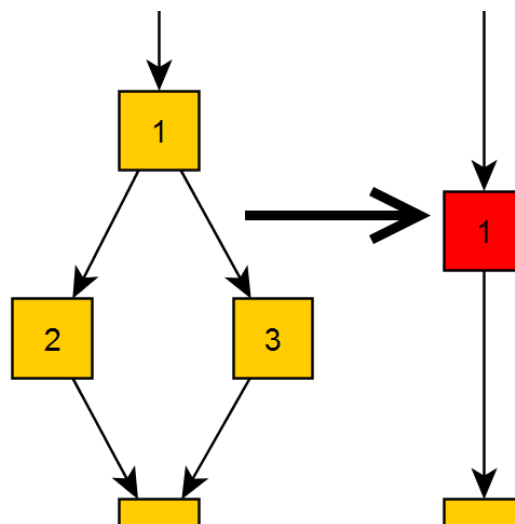


If-Then constructs can be reduced, because it can be shown, that if control enters in Block1, it will always continue in the after block, no matter if Block2 is traveled.

Requirements:

- Block1 has 2 children
- Block2 has only Block1 as parent
- Block2's child and Block1's other child is the same block (if any)
- The after block isn't Block1

- Reducing “If-Then-Else” constructs

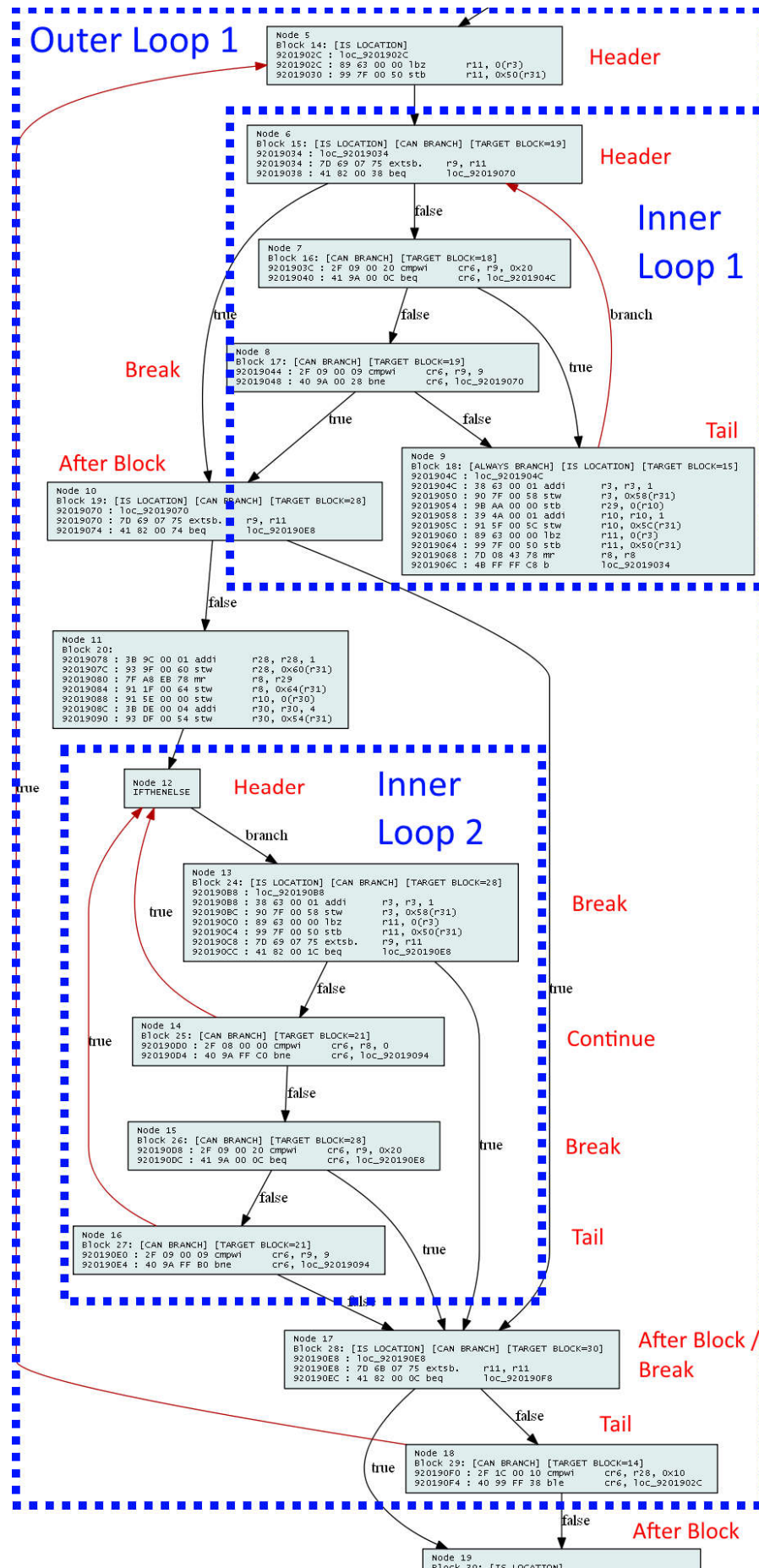


If-Then-Else constructs can be reduced like the If-Then constructs, just 3 nodes are reduced here.

Requirements:

- Block1 has 2 children
- Block2 and Block3 have Block1 as only parent
- Block2 and Block3 have the same only children (if any)

- Reducing loop parts and loops



To reduce cyclic regions back to loops, multiple preparation steps have to be taken.

1. Find cyclic regions in a given graph
2. For each region find out how many loops are contained
3. For each loop header find all possible simple looping paths
4. Combine looping paths to determine all associated nodes for a loop
5. Determine and reduce break and continue statements
6. Reduce the loop body graph to a single abstract node

1. In the first step an algorithm is used to determine so called “strongly connected components” (ref #4, #5). The result is a List of Regions, where each node can reach each other node via a cyclic path. If a region is not part of a cyclic component, it will only contain a single node index.

For the example above the algorithm results in:

```
0. SCCComponent = {0}
1. SCCComponent = {1}
2. SCCComponent = {21}
3. SCCComponent = {2}
4. SCCComponent = {3}
5. SCCComponent = {4}
6. SCCComponent = {5, 18, 17, 10, 6, 9, 7, 8, 13, 12, 11, 14, 16, 15}
7. SCCComponent = {19}
8. SCCComponent = {20}
9. SCCComponent = {22}
```

So in this graph there was only one such component found, and it contains all node indices of the outer loop from the example.

2. This is now used to find out how many loops exist in it, by counting the number of nodes that receive a “back edge” (ref #6, colored red in the example graph), so in this example that is Node 5, 6 and 12

3. To find all simple cycle paths in the region another algorithm is used: Johnson's Algorithm. (Ref #7, #8) For the example graph this results in:

```
Loop paths for component 6 :
0. Path = {5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18}
1. Path = {5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, 18}
2. Path = {5, 6, 7, 8, 10, 11, 12, 13, 17, 18}
3. Path = {5, 6, 7, 8, 10, 17, 18}
4. Path = {5, 6, 10, 11, 12, 13, 14, 15, 16, 17, 18}
5. Path = {5, 6, 10, 11, 12, 13, 14, 15, 17, 18}
6. Path = {5, 6, 10, 11, 12, 13, 17, 18}
7. Path = {5, 6, 10, 17, 18}
8. Path = {6, 7, 8, 9}
9. Path = {6, 7, 9}
10. Path = {13, 14, 12}
11. Path = {13, 14, 15, 16, 12}
```

Now for each loop header, all paths are grouped that include it and no other header node. As seen here, all paths for header node 5 contains other headers in its paths, so it cannot be reduced yet. But header nodes 6 and 12 have both 2 paths that contain no other header and so can be reduced in the next step.

4. To find all nodes associated with a loop, the path of the previous step are combined via union. So for this example that results in:

```
Simple Loops for component 6 :
0. Loop Header = 6, Loop Body = {7, 8, 9}
1. Loop Header = 12, Loop Body = {13, 14, 15, 16}
```

5. In order to determine and reduce break and continue statements these loop body nodes are now tested for:

- Does a child link back to the header node? -> Mark as continue node
- Is a child contained in the loop body set? -> Mark as not a tail node (longest possible cycle path)
- Is a child not contained in the body set? -> Mark as break, added target to list of exit nodes
- Is a parent not contained in the body set or header node? -> Mark loop as multi-entry, this cannot be reduced further without transformations

Now if a loop has more than one exit or an outside entry, the loop has to be skipped for now. The result for the example is:

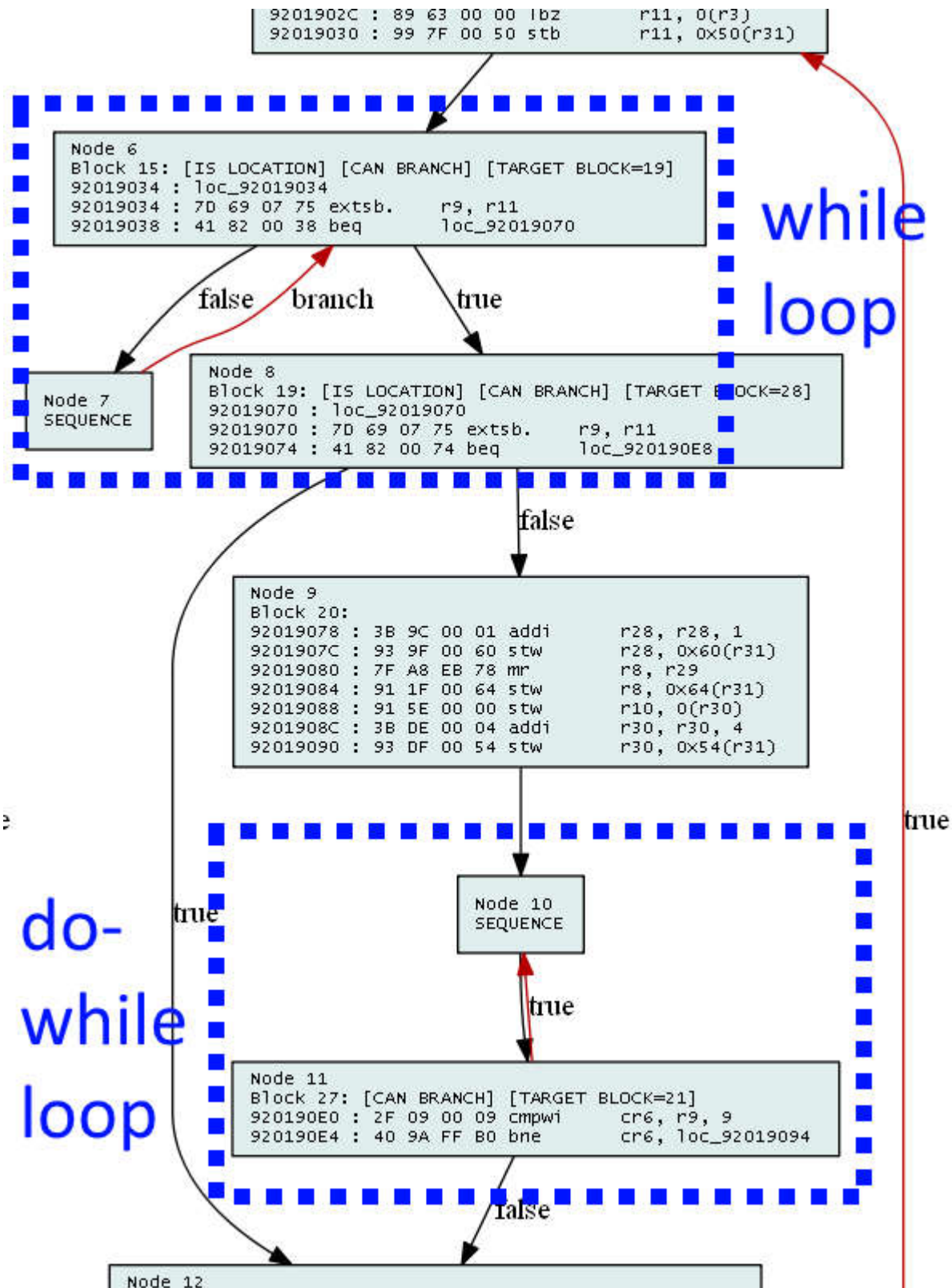
Simple Loop Information for component 6 :

```
0. Loop
  Header : Idx=6, IsBreak
  Body   : {
            Idx=7
            Idx=8, IsBreak
            Idx=9, IsTail
          }
  HasMultipleExits=False
1. Loop
  Header : Idx=12,
  Body   : {
            Idx=13, IsBreak
            Idx=14, IsContinue
            Idx=15, IsBreak
            Idx=16, IsTail, IsBreak
          }
  HasMultipleExits=False
```

This also allows later to determine what type of loop it is:

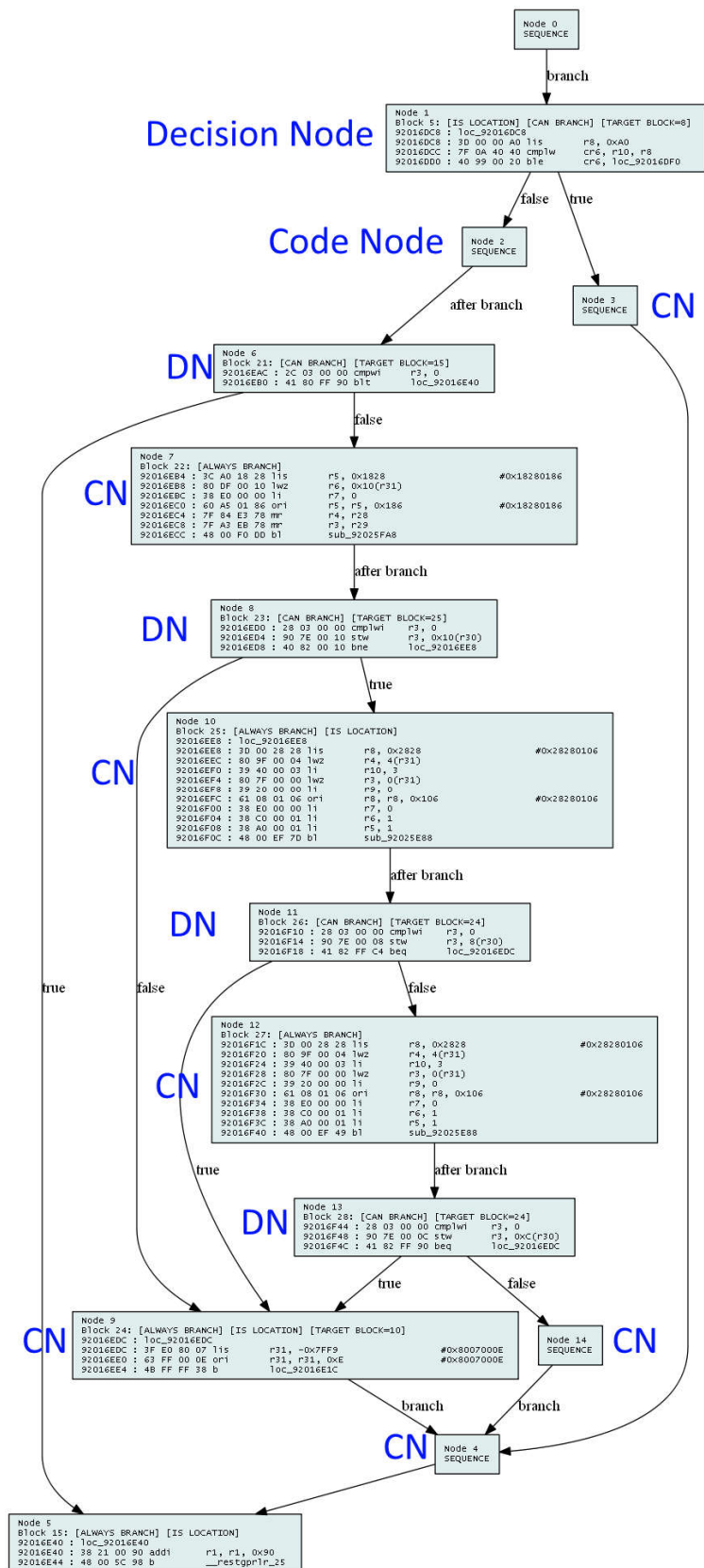
If the header node is also a break	<pre>while(header()) { body(); }</pre>
If the header node is not a break	<pre>do { header(); } while(body())</pre>

6. In order to reduce the body or header to a single node, nodes that are break or continue and not a tail node, get their outgoing or back going edges removed and are replaced with an abstract node, so it can later be reduced with a simple sequence reduction.



Now these loops can be reduced to a single abstract loop node for each loop, so that these also can be simplified later by sequence reductions for example. This in turn allows repeating this process with the outer loop that now contains no other loop headers anymore.

- Reducing Acyclic SESE Regions



link into the region and no link to the outside, all paths have to enter through the header node and leave through the exit node. For this the dominator and post dominator tree are created (ref #10). The post dominator tree is simply the dominator tree of the inverted graph. To find a SESE region a simple compare of all nodes with all other nodes is done, where its checked if the first node dominates the other and the second node post-dominates the first.

For this example the result is:

```
All SESE regions:
0, 1, 2, 6, 7, 8, 9, 4, 10, 11, 12, 13, 14, 3, 5
1, 2, 6, 7, 8, 9, 4, 10, 11, 12, 13, 14, 3, 5
```

Dominator Tree

```
+--0
| +-1
| | +-2
| | | +-6
| | | | +-7
| | | | | +-8
| | | | | | +-9
| | | | | | +-10
| | | | | | | +-11
| | | | | | | +-12
| | | | | | | +-13
| | | | | | | | +-14
| | +-3
| | +-4
| | +-5
```

Post Dominator Tree

```
+--5
| +-4
| | +-14
| | +-13
| | | +-12
| | | +-11
| | | +-10
| | +-9
| | +-8
| | | +-7
| | +-3
| +-6
| | +-2
| +-1
| | +-0
```

Like with the loop-reduction, regions that contain other regions are skipped for now until those inner regions got reduced. This is determined via list intersection.

Once such a region has been found, the region's graph is cut out and used as data for the abstract node that will replace it. The inner structure will later processed further when the AST gets compiled out again. It is there for good to only allow such reductions once none of the others are possible anymore, so no inner reductions were left to do.

4. The Tree Problem – getting readable code back

If previous reduction steps succeeded in reducing the graph to a single end node, then this will contain either a simple code block or an abstract block. As these are only sequential or tree structures, it can be seen as an abstract syntax tree, short AST.

At this point, a recursive function just has to print the different types out and add tabs depending on the depth.

One-block-while, sequences, if/then, if/then/else, while and do-while blocks are easy to express and are not more detailed explained in this paper.

For the acyclic SESE regions however some further processing is needed, as it has a graph as data and not a tree.

Now inside such a region, nodes can be marked by either being a decision node (having more than one child) or being a code node (having only one child) and the idea is, that for each code node, there exist one or multiple paths to reach them. Each path has to go through decision nodes and the edge it has to travel decides the condition in the resulting boolean expression. Then all paths can be summed up by OR'ing the resulting expressions.

For this example all reaching paths for all nodes results in:

```
Reaching paths for Node 1 :
  0->1
Reaching paths for Node 2 :
  0->1->2
Reaching paths for Node 3 :
  0->1->3
Reaching paths for Node 4 :
  0->1->2->6->7->8->9->4
  0->1->2->6->7->8->10->11->9->4
  0->1->2->6->7->8->10->11->12->13->9->4
  0->1->2->6->7->8->10->11->12->13->14->4
  0->1->3->4
Reaching paths for Node 5 :
  0->1->2->6->5
  0->1->2->6->7->8->9->4->5
  0->1->2->6->7->8->10->11->9->4->5
  0->1->2->6->7->8->10->11->12->13->9->4->5
  0->1->2->6->7->8->10->11->12->13->14->4->5
  0->1->3->4->5
Reaching paths for Node 6 :
  0->1->2->6
Reaching paths for Node 7 :
  0->1->2->6->7
Reaching paths for Node 8 :
  0->1->2->6->7->8
Reaching paths for Node 9 :
  0->1->2->6->7->8->9
  0->1->2->6->7->8->10->11->9
  0->1->2->6->7->8->10->11->12->13->9
Reaching paths for Node 10 :
  0->1->2->6->7->8->10
Reaching paths for Node 11 :
  0->1->2->6->7->8->10->11
Reaching paths for Node 12 :
  0->1->2->6->7->8->10->11->12
Reaching paths for Node 13 :
  0->1->2->6->7->8->10->11->12->13
Reaching paths for Node 14 :
  0->1->2->6->7->8->10->11->12->13->14
```

Also applying topological order gives the order in which each code node can be first reached (ref #11, #12), so that for each codeblock if statements can be prepared and then printed out in the order they would have been executed.

Topological Order = {0, 1, 2, 3, 6, 7, 8, 10, 11, 12, 13, 9, 14, 4, 5}

The result for combining the condition paths for each code node and then iterating them out is: (please note that the example graph had already simple reductions, so the node ids don't match here and I marked the beginning instead)

```
void sub_92016D78()
{
    Block0();
    if(Block1())
    {
        if(!Block3())
        {
            Block4();
        }
    }
    else
    {
        Block2();
    }
    if(!Block5()) //Region start, first if statement with condition path
    {
        if(Block6()) //This code node was an abstract one, so it has internal structure
        {
            if(!Block16())
            {
                if(!Block17())
                {
                    Block18();
                }
            }
        }
        else
        {
            Block7();
        }
        Block19();
        Block20();
    }
    if(!Block5() && !Block21()) //Second if statement
    {
        Block22(); //Second code node, this time not abstract
    }
    if(!Block5() && !Block21() && Block23()) //Third if statement
    {
        Block25(); //Third code node
    }
    if(!Block5() && !Block21() && Block23() && !Block26()) //...
    {
        Block27(); //...
    }
    if((!Block5() && !Block21() && !Block23()) ||
        (!Block5() && !Block21() && Block23() && Block26()) ||
        (!Block5() && !Block21() && Block23() && !Block26() && Block28()))
    {
        Block24();
    }
    if(!Block5() && !Block21() && Block23() && !Block26() && !Block28())
    {
        Block29();
        Block30();
    }
    if(Block5())
    {
        Block8();
        Block9();
    }
}
```

```

    if ((!Block5() && !Block21() && !Block23()) ||
        (!Block5() && !Block21() && Block23() && Block26()) ||
        (!Block5() && !Block21() && Block23() && !Block26() && Block28()) ||
        (!Block5() && !Block21() && Block23() && !Block26() && !Block28()) ||
        (Block5()))
    {
        if (!Block10())
        {
            if (!Block11())
            {
                Block12();
                Block13();
            }
        }
        Block14();
    }
    Block15();
}

```

Obviously this results in the reusage of decision nodes. This can be bypassed by evaluating the decision nodes once at the beginning and store the result in temporary variables. Like “c1=Node1(); c2=Node2();...” and then “if(c1 && !c2){...}”. Also code nodes that have the same minimal reaching path and are executed after each other, could be grouped into an if statement containing the initial, minimal path and create sub if statements for the rest of the paths.

5. Acknowledgements

First I would like to thank Pavel for making me learn all this so I can help him.

Then I would like to thank xorlooser for his ida pro plugin to read xbox 360 xex files and allowing me to get proper asm output from IDA.

Thanks to the IDA Pro and Hexrays teams, finally I roughly understand how they work

I also would like to thank Tushar Roy for his excellent tutorial videos on graph algorithm

And of course I have to say thanks to the authors of the “no more gotos” paper, it was a very good guide on the way

6. References

#	Reference	Description
1	https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/yakdan/dream_ndss2015.pdf	Paper “No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations”
2	https://github.com/zeroKilo/DirectedGraphsWV/tree/master/XEXDecompiler3	XEX Decompiler 3
3	https://en.wikipedia.org/wiki/Tree_traversal#Post-order	Post Order Traversal
4	https://en.wikipedia.org/wiki/Strongly_connected_component	Strongly connected components
5	https://www.youtube.com/watch?v=RpgcYiky7uw	“Kosaraju's Algorithm”
6	https://courses.csail.mit.edu/6.006/fall11/rec/rec14.pdf	“DFS Edge Classification”, depth first search to find back edges
7	https://www.youtube.com/watch?v=johyrWospv0	“Johnson's Algorithm - All simple cycles in directed graph”
8	https://en.wikipedia.org/wiki/Johnson%27s_algorithm	Johnson's Algorithm
9	https://en.wikipedia.org/wiki/Single-entry_single-exit	SESE Region
10	https://en.wikipedia.org/wiki/Dominator_(graph_theory)	Dominator Tree
11	https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf	Topological Sort
12	https://www.youtube.com/watch?v=ddTC4Zovtbc	Topological Sort Graph Algorithm