Thesis

Gilles De Borger

April 1, 2021

1 Introduction

I am writing a tool to mitigate Nemesis attacks. To mitigate attacks a specific condition has to hold, that for any i, the i'th instruction in the if and in the else branch have the same latency. In order to ensure this property holds I instrument the binary code with new instructions to ensure that this property holds for all secret dependent branches without modifying the program execution (i.e. all instruction should have no effect)

2 Implementation

2.1 Introduction

The tool instruments a program's binary by manipulating the program's Control Flow Graph (CFG). The CFG is a datasructure constructed from the binary in such a way as to allow reconstructing the binary from the CFG later on.

2.2 control flow graph

The Control Flow Graph (CFG) is a data structure that represents the control flow of a program. A CFG consists of nodes V and directed edges E. Each node V contains a contiguous sequence of instructions. Associated with each node in V is a sequence of latencies equal to the latencies of the node's instructions. An edge is drawn from node v to node v' if and only if the last instruction in v can be followed by the first instruction in v' when following program control flow.

Two nodes are said to be balanced if they have the same number of instructions and if they corresponding instruction has the same latency. This is equivalent to saying that their latency sequences are the same.

A node is said to be at depth n if there is a path from the root to the node of length n. Because it is possible that there are multiple paths to a node of different lengths a node can be at multiple depths.

A node satisfies the Nemesis property if for each pairs of descendants at depth i it holds that the nodes are balanced.

2.3 Balancing a tree-like CFG

Ensuring the Nemesis property holds for a given node is relatively straightforward under certain conditions. If the CFG has a tree-like structure a recursive strategy is possible to ensure the Nemesis property holds for a given node. A CFG is said to have a tree-like structure when all nodes are at exactly one depth.

If the CFG has a tree-like structure a recursive strategy can be applied to ensure the Nemesis property holds for any given node; to ensure the Nemesis property holds first ensure it holds for its descendants.

The basic case of this recursive procedure is is to balance two leaves (i.e. to balance two nodes). Figure 1 illustrates how this is done. First a target latency sequence is determined. This sequence is calculated from the latency sequences of the nodes that are being balanced. Implicit in this calculation are the constraints that are imposed on inserting instructions into a program. In some cases it is not possible to insert a single instruction without altering the program state. In these cases additional instructions need to be inserted to store and restore the program state. This inserting of additional instructions is reflected in the calculated

```
def balance_nodes(n1, n2):
  target_latencies = balance_node_latencies(n1.latencies, n2.latencies)

  new_instructions_n1 = create_new_instruction_sequence(n1.instructions, target_latencies)
  n1.replace_instructions(new_instructions_n1)

  new_instructions_n2 = create_new_instruction_sequence(n1.instructions, target_latencies)
  n2.replace_instructions(new_instructions_n2)
```

Figure 1: node balancing algorithm

target latency sequence. This ensure that in the following step it is possible to generate a new instruction sequence such that the following conditions hold:

1. all instructions that are present in the original sequence are also present in the new sequence
2. the latencies of the new instruction sequence are equal to the target latencies
3. the original and the new instruction sequence have the same effects on the program state and output

Once these instruction sequences have been generated it is trivial to replace the original instruction sequences.

The recursive procedure for ensuring the Nemesis property holds for a given node is as follows. First ensure it holds for its immediate successors by balancing their descendants. To balance both of the subtrees, first balance the roots of the subtrees. Then determine for each depth i below the target node a target latency sequence. Because both of the subtrees are balanced they will each have one well defined latency sequence at each depth i. The target latency sequence can be calculated in the same way as it was calculated for individual nodes. Then for each node at depth i, generate a new instruction sequence and insert it into the node. This process is illustrated in figure 2.

2.4 Balancing non-tree-like CFG

The recursive strategy is not applicable to all control flow graphs. Figure 3 illustrates two cases where the it is not possibly to recursively balance nodes. The firs case arises when some sequence of instructions is only executed if some condition is true. the second case arises when the control flow jumps to an earlier instruction. This is the case anytime a loop is used inside a program.

In the case of a conditional node the issue stems from the fact that there are paths of different lengths going to a node (the node labeled '1' in the example). Such nodes are by definition both at a depth n and a depth $n+k$, where k depends on the difference between the path lengths. The following node will be at depth $n+1$ and $n+k+1$ and so. This increases the number of nodes that need to be balanced with respect to each other. As is the case in the example, each node below node '3' will have to be balanced with its immediate successor. This is undesirable because balancing is commutative. The node '3' would have to be balanced with all of its descendants.

To prevent this from happening empty nodes are inserted into the graph to ensure that all paths to any given node are always of the same length. The balancing operation for two nodes is well defined even if one of the nodes is empty so this poses no issues.

Loops in the control flow graph make it impossible to apply a recursive algorithm, since the recursion will go on indefinitely. Additionally any node in a loop will be at infinitely many depths, since there are infinitely many paths to it, which introduces the same aforementioned problems. In order to balance such a CFG the graph is first unwound, removing any cycles present. For a given loop, let the first node in the loop be the node closest to the root, and the last node the node furthest from the root. A new node is created identical to the first node in the loop. The edge that connects the last and the first edge is removed and replaced by a new edge from the last node to the newly created copy of the first node. Figure 4 illustrates how this works. After the graph has been balance the same procedure is executed in reverse in order to recreate the cycle.

During balancing, any operation performed on the original node is also performed on the copy to ensure they are identical at all times.

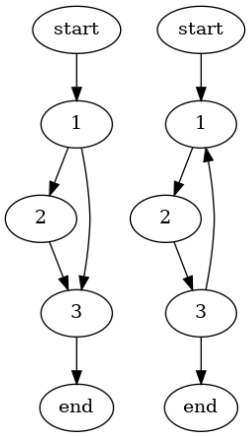
```
def balance_trees(tree1, tree2):

  balance_branching_point(tree1)
  balance_branching_point(tree2)

  target_sequences = balance_tree_latencies(tree1, tree2)

  for each depth d:
    for each node v at depth d:
      new_instructions =     create_new_instruction_sequence(v.instructions,
  target_sequences[d])
        v.replace_instructions(new_instructions)
```

Figure 2: tree balancing algorithm



(a) conditional (b) loop

Figure 3: problem CFGS

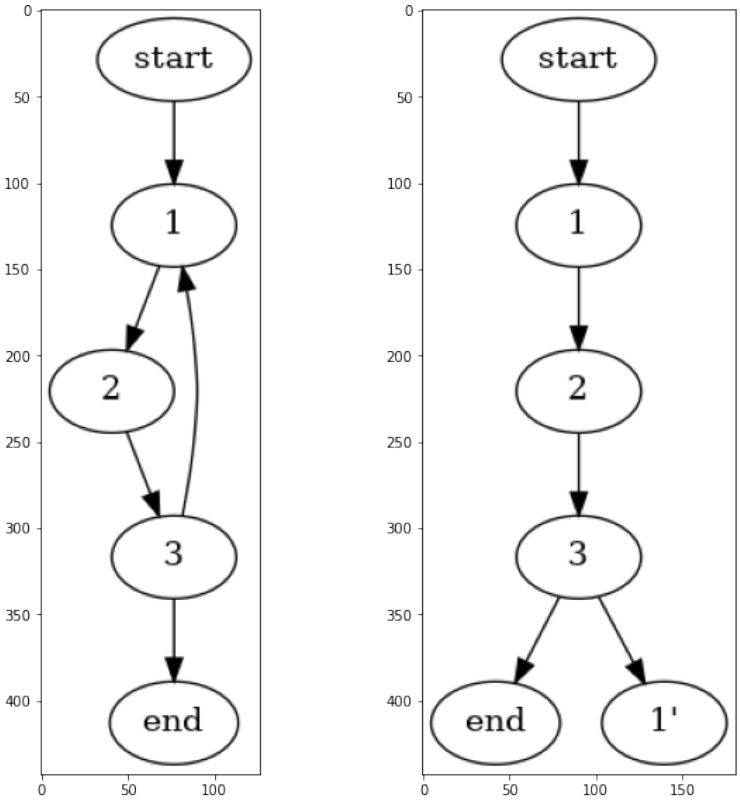


Figure 4: unwinding cycles