KU LEUVEN
FACULTEIT
INGENIEURSWETENSCHAPPEN

# The best master’s thesis ever

Gilles De Borger

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Artificiële intelligentie

**Promotor:**
Prof. dr. ir. Knows Better

**Assessoren:**
Ir. Kn. Owsmuch
K. Nowsrest

**Begeleiders:**
Ir. An Assistent
A. Friend

Academiejaar 2020 – 2021

# Preface

PLACEHOLDER I would like to thank everybody who kept me busy the last year, especially my promoter and my assistants. I would also like to thank the jury for reading the text. My sincere gratitude also goes to my wive and the rest of my family. PLACEHOLDER

*Gilles De Borger*

# Contents

# Abstract

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.s "Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?"

# Samenvatting

In dit `abstract` environment wordt een al dan niet uitgebreide Nederlandse samenvatting van het werk gegeven. Wanneer de tekst voor een Nederlandstalige master in het Engels wordt geschreven, wordt hier normaal een uitgebreide samenvatting verwacht, bijvoorbeeld een tiental bladzijden.

# List of Figures and Tables

## List of Figures

## List of Tables

# Chapter 1

# Introduction

This is the introduction to the text ...

## 1.1 First topic of the Chapter

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

## 1.2 Second topic of the chapter

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

# Chapter 2

# Design

## 2.1 Introduction

The root cause of the vulnerability exposed by Nemesis-style attacks are differences in the latencies of two instruction that occur at the same position in two branches of a secret-dependent branching instruction. In practice an attacker can exploit this vulnerability by generating latency traces along different paths of the program's control flow. Any differences in instruction latencies will be reflected as differences in these latency traces. By carefully inspecting the relevant sections of the latency traces the attacker can infer which paths were taken for a given input. In cases where the path depends on some secret data the attacker is then able to infer information about this data, successfully leaking information.

The goal of the algorithm outlined in this section is to ensure that latency traces cannot be used to leak information in this way. It does this by inserting additional instructions into branches of a secret-dependent branching instruction. These instructions are carefully selected such that they have the same latency as their corresponding instruction in the other branch. This ensures that any instructions that occur at the same position in two different branches have the same latency. As a result the sections of latency traces that correspond to these branches will be identical, making it impossible for an attacker to infer information.

The proposed algorithm inserts additional instructions into the program through manipulation of the program's control flow graph. This graph consists of nodes and vertices, where each node contains a sequence of instructions. One of the main operations performed on the graph is the alignment of a set of nodes. This operation inserts additional instructions into nodes such that all instructions at a given position in any of the nodes have the same latency.

Not all structures found in a control flow graph are suitable for alignment. The aforementioned alignment operation therefore has some conditions on the structure of the control flow graph that need to be met. The other main operation of the algorithm therefore consists of inserting additional nodes into the graph such that these conditions are met.

Section 2.2 will formally define the property that needs to hold for a program in order for Nemesis-style attacks to be mitigated. Section 2.3 introduces the CFG data structure and translates the aforementioned property to such structures. Finally, sections 2.4 and 2.5 describe the insertion and alignment of nodes, respectively.

## 2.2 Nemesis-sensitive property

In their paper Pouyanrad et. al have formally defined the Nemesis-Sensitive property. Let $region^{then}(ep)$ and $region^{else}(ep)$ capture the set of execution points belonging to the branch target and the other region of some branching instruction $ep$. Let $ep^i$ be the i'th instruction in a region. A program P with a secret-dependence branch in $ep$ and $region^{then}(ep)$ and region $region^{else}(ep)$ with the same number of execution points, satisfies the nemesis-sensitive property if and only if:

$$\begin{aligned} \forall ep^i \in region^{then}(ep) : \forall ep^j \in region^{else}(ep) \ such\ that\ i = j : \\ (s_{ep^i} \xrightarrow{t} s_{ep^i_{next}}) \wedge (s_{ep^j} \xrightarrow{t'} s_{ep^j_{next}}) \iff t = t' \end{aligned} \tag{2.1}$$

[7]

The relation $s \xrightarrow{t} s'$ models the transition between program states $s$ and $s'$, declaring that the transition between $s$ and $s'$ takes a time $t$. For a given instruction this time $t$ is equal the instruction's latency. This property states that for any two corresponding instructions in the branches their latencies should be the same.

If this nemesis-sensitive property holds for a program then an attacker is not able to infer which branch was taken by the program based on instruction latencies.

## 2.3 CFG

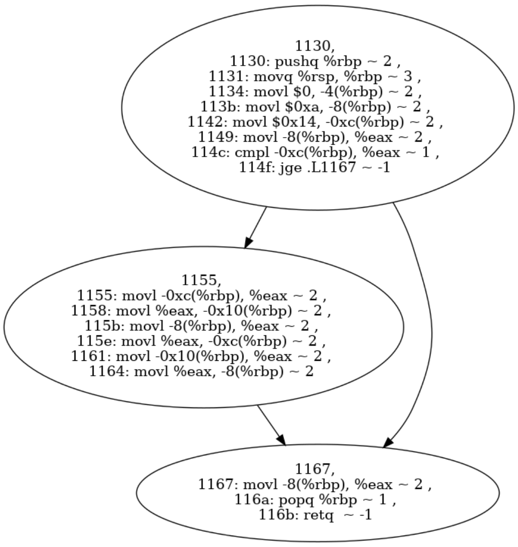
The Control Flow Graph (CFG) is a data structure that represents the control flow of a program. A CFG consists of nodes $V$ and directed edges $E$. Each node $V$ contains a contiguous sequence of instructions that is always executed as a whole. This implies that any branching instruction can only occur at the end of a node, and an instruction that is the target of a branching sequence can only occur at the start.

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. The algorithm only considers branching instructions that are binary in nature, so a node in the CFG can have at most 2 successors. By construction of this data structure a branching instruction will always be the last instruction in a node. A node is said to be secret-dependent if its last instructions is a secret-dependent branching instruction.

Each node has a latency sequence associated with it, equal to the latencies of the node's instructions. A latency trace along a path of the CFG is then equal to the concatenation of the latency sequences of each node along the path. Figure 2.1 shows an example of a such a CFG, along with the original program it is created from. The

```
int main(){
        int a = 10;
        int b = 20;
        if (a < b){
            int temp = b;
            b = a;
            a = temp;
        }
        return a;
}
```

(a) C program



```
1130,
1130: pushq %rbp ~ 2 ,
1131: movq %rsp, %rbp ~ 3 ,
1134: movl $0, -4(%rbp) ~ 2 ,
113b: movl $0xa, -8(%rbp) ~ 2 ,
1142: movl $0x14, -0xc(%rbp) ~ 2 ,
1149: movl -8(%rbp), %eax ~ 2 ,
114c: cmpl -0xc(%rbp), %eax ~ 1 ,
114f: jge .L1167 ~ -1
```

```
1155,
1155: movl -0xc(%rbp), %eax ~ 2 ,
1158: movl %eax, -0x10(%rbp) ~ 2 ,
115b: movl -8(%rbp), %eax ~ 2 ,
115e: movl %eax, -0xc(%rbp) ~ 2 ,
1161: movl -0x10(%rbp), %eax ~ 2 ,
1164: movl %eax, -8(%rbp) ~ 2
```

```
1167,
1167: movl -8(%rbp), %eax ~ 2 ,
116a: popq %rbp ~ 1 ,
116b: retq  ~ -1
```

(b) Corresponding CFG

Figure 2.1: Example program with corresponding CFG

CFG also contains the latency for each instruction. Note that by convention the only node with no incoming edges is considered the starting node of the CFG.

Following the property described in section 2.1, the nemesis-sensitive property can be defined for a node in the CFG. Let $v$ be a secret-dependent node. Let $v_f$ be a node such that all paths from $v$ to some leaf go through $v_f$. Then $region^{then}(v)$ can be defined as the set of nodes reachable following the first of $v$'s outgoing edges up to and including $v_f$ and $region^{else}(v)$ as the set of nodes reachable following the other outgoing edge up to and including $v_f$.

Any differences in the latency sequences of two nodes can only be used to infer which branch was taken at the nearest branching point that is an ancestor of both nodes. Any differences in latencies between two nodes that are descendants of $v_f$ can therefore only be used to infer information about which branch was taken at $v_f$. This means that all nodes below $v_f$ do not have to be considered. If no such node $v_f$ exists then the regions simply consists of all nodes reachable from $v$ through one of its outgoing edges.

Let $n^i \in region(v)$ be a node such that there is a path going to it from node $v$ of length $i$. The depth of $region(v)$ is defined as being the length of the longest path from $v$ to some node $v' \in region(v)$ that does not contain a cycle.

A secret-dependent node $v$ and $region^{then}(v)$ and $region^{else}(v)$ with the same depth satisfies the nemesis-sensitive property if and only if

$$\begin{aligned}\forall n^i \in region^{then}(v) : \forall n^j \in region^{else}(v) \; such\ that\ i = j :\\ latencies(n^i) = latencies(n^j)\end{aligned} \tag{2.2}$$

where $latencies(n)$ is a function mapping a node $n$ to its latency sequence. This property states that the latency sequence of any two nodes that are the same distance

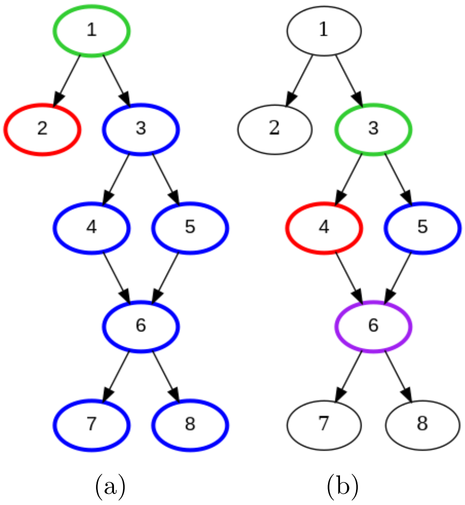


Figure 2.3: then-else regions for secret-dependent nodes

away from some secret-dependent node need to have identical latency sequences. If this property holds then the critical sections of latency traces will be identical and cannot be used to infer information about the secret-dependent branch.

Figure 2.3 illustrates how the borders of each region are defined. The secret-dependent node is marked in green, while the two branches are marked in red and blue. In the second example, the node marked in purple belongs to both regions. In example 2.2a there is node node such that all paths from the secret-dependent node to a leaf go through it, so the regions extend all the way to the leaves. In example 2.2b all paths that start in the secret-dependent node go through the node 6. Any differences in nodes 7 and 8 can only be used to infer information about the branch in node 6. These nodes therefore do not have to be considered.

## 2.4 Equalising

There are 2 structures found in a program's control flow graph that make it impossible to enforce the nemesis-sensitive property for a node as defined in the previous section. These structures are shown in figure 2.5. The first stage of the algorithm consists of inserting nodes such that these structures no longer occur.

The first such structure occurs when the program contains some sequence of instructions that is only executed if some condition is true and is illustrated in fig . In such a CFG there will be some node that has two paths to it from the root where the paths have different lengths. One path will contain the node that corresponds to the conditional instructions, while the other path will not.

In such cases it is impossible to equalize the latency traces of the two paths. Because the paths have different lengths one of the latency traces will always be longer than the other one. Additionally, because all nodes in the shorter path are also nodes in the longer path it is impossible to modify the shorter path without also modifying the longer path. Any attempts to lengthen the shorter path therefore also lengthen the longer path. If these paths start at a secret-dependent node it is impossible to ensure that the nemesis-sensitive property holds.

The second problematic structure occurs when one of the branches is shorter than the other one, as shown in figure . In such cases there will be some nodes in one branch that have no corresponding nodes in the other branch, making it impossible to align them.

The nemesis-sensitive property as defined in section 2.2 entails that it is impossible for a node to satisfy the property if one of these structures occurs in its branches, since in both cases the regions have different depths. The first stage of the algorithm therefore consists of first equalizing all path lengths and then equalizing branches. Algorithms 1 and 2 depict pseudo-code for equalizing paths lengths and equalizing branches respectively.

### 2.4.1 Extract Sub-graph

The different procedures described in this section only need to take into account the branches of secret dependent nodes. These branches correspond to the regions $region_{then}(v)$ and $region_{else}(v)$ as defined in section 2.3. The procedure *ExtractSubgraph*, shown in 1, extracts the subset of the graph that contains only the nodes that belong to either one of these regions for a given secret-dependent node. The edges of this new CFG are all the edges of the original CFG whose head and tail are a part of this subset.

To determine which nodes are a part of this subgraph all immediate dominators are determined starting from node $n$. A node $u$ is said to dominate another node $w$ with respect to $n$ if every path from $n$ to $w$ passes through $u$. Node $v$ is the immediate dominator of $w$ if $v$ dominates $w$ and every other dominator of $w$ dominates $v$ [6]. The immediate dominator is determined for each node reachable from $n$. If all leaves reachable from $n$ have the same immediate dominator $d$ then all paths from $n$ to some leaf go through $d$. In this case any descendants of $d$ are not part of $region_{then}(v)$ or $region_{else}(v)$ and do not have to be included in the sub-graph.

If such a node $d$ exists then the nodes that are a part of the sub-graph are all nodes that are on a path from $n$ to $d$. If $d$ does not exists the sub-graph nodes are all nodes that are on a path from $n$ to some leaf. This definition is analogous to the definition for $region_{else}(v)$ and $region_{then}(v)$ as defined in section 2.3.

### 2.4.2 Equalize paths

To equalize all path lengths starting from some secret-dependent node $v$, first a subset of the graph's nodes are extracted such that only the regions $region^{then}(v)$ and $region^{else}(v)$ are considered. Next the length of the longest path is computed from $v$ to all nodes in the sub-graph.

Let $(u, v)$ be an edge in the sub-graph. Let $d(u)$ and $d(v)$ be the length of the longest path to $u$ and $v$. If the difference between $d(u)$ and $d(v)$ is more than one then there exists at least two paths to $v$. The first path goes through $u$ and has length $d(u) + 1$. The second path goes through a different predecessors of $d(v)$ and has length $d(v)$.

The procedure for equalizing the path lengths iterates over all edges of the sub-graph. If the distances to $u$ and $v$ differ by more than one then nodes are inserted into the edge between $u$ and $v$ such that the path that goes to $v$ through $u$ is of the same length as the longest path to $v$.

### 2.4.3 Equalize branches

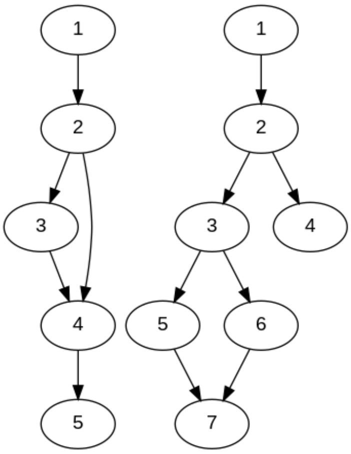
The branches of the CFG can be equalized in a similar way. Given some secret-dependent node $v$ a subset of the graph's nodes are extracted. The lengths of the longest paths are computed for all nodes in the sub-graph. The maximum path length is then determined as being the longest path length to some node that is also leaf. The procedure then iterates over all leaves in the sub-graph and determines the difference between the distance to the leaf and the maximum path length. If this difference is larger than zero then additional nodes are inserted as the predecessor to the leaf until the distance to the leaf is equal to the maximum path length. Because the final instruction in a leaf is a return statement any new nodes have to be added as predecessors.

### 2.4.4 Cycles

**Is this section necessary? technically cycles aren't an issue when it comes to equalizing paths and branches, they are removed as part of my concrete implementation** The aforementioned operations do not work if there are cycles in the CFG. Before equalizing paths and branches all cycles are removed from the CFG. To remove a cycle the depth for each node in the cycle is determined. The depth of a node is equal to the length of the longest path to the node. The edge that connects the deepest node to the most shallow node is removed from the graph to make the graph acyclic. Once all paths and branches are equalize the edge can then be re-inserted to restore the cycle.

## 2.5 Alignment

During the alignment stage the nodes of the CFG are aligned in a level-wise manner. The alignment of a set of nodes consists of inserting instructions such that all instructions at a given position across all nodes in the set have the same latency. The level of a node is defined as being the distance between the root of the graph and the node. The first stage of the algorithm ensures that all paths to a given nodes have the same length making the level of a node a well defined value. The alignment stage iterates over all the levels of the sub-graph and aligns the set of nodes found at that level. Algorithm 3 depicts pseudocode for this stage of the algorithm.



(a) optional node (b) unequal branches

Figure 2.5: problematic structures in CFG

### 2.5.1 Basic Operation

The core of the alignment operation consists of repeatedly selecting a reference node and inserting instructions into the other nodes to match the latencies of the reference node. Each iteration a set of candidate nodes is determined, from which the reference node is then selected.

An index variable $i_{ref}$ is used to keep track of the position of the first instruction that has not yet been aligned. This variable is initially equal to zero and is incremented every iteration. The instruction at position $i_{ref}$ in the reference node is called the reference instruction.

The algorithm iterates over all nodes that are not the reference node and verifies if the instruction at position $i_{ref}$ has the same latency. If the two latencies are not equal or if the node is shorter than the reference node a new instruction is inserted at position$i_{ref}$. The latency of this new instruction is equal to the latency of the reference instruction. Once this has been repeated for all nodes in the set then all instruction with at position $i_{ref}$ have the same latency and the variable can be incremented.

### 2.5.2 Selecting the Reference Node

Because an instruction is potentially added to each node that is not the reference node the reference node needs to have at least as many instructions as the node with the largest number of instructions. This ensure that at some point all nodes have the same number of instructions. The set of candidate nodes therefore consists of all nodes that have $n_{max}$ instructions, where $n_{max}$ is the number of instructions in the longest node.

The selection of the reference node also determines the reference instruction. If there are any instructions at position $i_{ref}$ that are not a branching or return instruction then the reference instruction cannot be a branching or return instruction. Branching instructions and return instructions are special cases that will result in the insertion of a branching instruction. If there is a regular instruction in some node at position $i_{ref}$ then inserting a branching instruction can lead to errors in the control flow of the program.

The reference node has to be selected such this situation cannot occur. From the set of candidates only those nodes are considered that do not have a branching instruction at position $i_{ref}$. From this subset a reference node can be selected arbitrarily. If among the candidates all nodes have a branching instruction at $i_{ref}$ then it is safe to insert a branching instruction and a reference node can also be selected arbitrarily.

### 2.5.3 Constructing NOP instruction

For each latency class a template NOP instruction has been determined. The instruction can be inserted into the program as-is if it has no effect on the program state, i.e. it does not modify any register values. If the instruction does modify some register the algorithm selects a registers that can safely be used. This needs to be a register that is not in use at the time of execution of the instruction.

There are two types of free registers. A register can be free because its current value is no longer used. This occurs when the register is overwritten at some later point without being read first. Alternatively a register can be free because it isn't used anywhere in the current function. In the latter case, however, it is possible that the register is in use by the caller, since there is no guarantee that the caller stored all the registers it uses.

The function is statically analyzed to determine which registers are free to use for this purpose. If a register of the first type exists then it can be used as the operand of the NOP instruction and the resulting instruction can be inserted as-is into the node. If no such registers exists, a free register of the second type is selected. In this case additional instructions are inserted into the program to ensure that the original value of the register is not lost. In the root of the CFG additional instructions are inserted to push the register value onto the stack, while in every leaf instructions are inserted that pop the value from the stack. Once these instructions have been inserted the register effectively becomes a free register of the first type and can later be reused when construction additional NOP instructions.

If there are no free registers available, any register is arbitrarily selected. Additional instructions are inserted before and after the NOP instruction to push and pop the register value. To ensure the nodes are still balanced these push and pop instructions are inserted across all nodes of the current level.

If the reference instruction is a branching instruction the the NOP instruction will also be a branching instruction. The target of the branching instruction is then the address of the first instruction of the node's successors.

If the reference instruction is a call to a function then the NOP instruction will be a function call to the same function.

**Algorithm 1:** Equalize Path Lengths

```
Procedure EqualizePathLengths(g: CFG, v: Node)
    subgraph ← ExtractSubGraph (g, v)
    longestPathLengths ← ComputeLongestPathLengths(subgraph, v)
    forall (u, v) ∈ Edges(subgraph) do
        diff ← longestPathLengths[u] - longestPathLengths[v]
        if diff > 1 then
            head ← v
            for i ∈ 1, 2, ..., diff-1 do
                newNode ← CreateNode()
                InsertNodeBetween(newNode, u, head)
                head ← newNode
            end
    end
Function ComputeLongestPathLengths(g: CFG, s: Node)
    dist = {n : -1 | n ∈ Nodes(g) }
    dist[s] ← 0
    forall n ∈ TopologicalOrder(g) do
        forall succ ∈ Successors(n) do
            dist[succ] ← Max(dist[succ], dist[n] + 1)
        end
    end
    return dist
Function ExtractSubGraph(g: CFG, n: Node)
    immedidateDominators ← computeImmediateDominators(g, n)
    leafDominators ← { u ∈ Nodes(g) |$(u, v)$ ∈ leafDominators ∧v ∈
     Leaves(g) }
    if | leafDominators | = 1 then
        dominator ← leafDominators[0]
    else
        dominator ← ∅
    subgraphNodes ← { }
    if domintor ≠ ∅ then
        forall path ∈ computeAllPaths(g, n, dominator) do
            $subgraphNodes \leftarrow subgraphNodes \cup (\{p|p \in$
             $path\} \setminus subraphNodes)$
        end
    else
        forall l ∈ Leaves(g) do
            forall path ∈ computeAllPaths(g, l, dominator) do
                $subgraphNodes \leftarrow subgraphNodes \cup (\{p|p \in$
                 $path\} \setminus subraphNodes)$
            end
        end
    $subgraphEdges \leftarrow \{(u, v)|u \in subgraphNodes \wedge v \in$
     $subgraphNodes \wedge (u, v) \in$ Edges(g) }
    return $(subgraphNodes, subgraphEdges)$
```

**Algorithm 2:** Equalize Branches

```
Procedure EqualizeBranches(g: CFG, n: Node)
    subgraph ← ExtractSubGraph (g, v)
    longestPathLengths ← ComputeLongestPathLengths(subgraph, v)
    maxPathLength ←
     Max({longestPathLengths[v] | v ∈ Leaves(subgraph)})
    forall leaf ∈ Leaves(subgraph) do
        diff ← longestPathLengths[leaf] - maxPathLength
        if diff > 0 then
            v ← leaf
            for i ∈ 1, 2, ..., diff do
                newNode ← CreateNode()
                AddNode(g, newNode)
                forall p ∈ Predecessors(v) do
                    AddEdge(g, (p, newNode))
                    RemoveEdge(g, (p, v))
                end
                AddEdge(g, (newNode, v))
            end
    end
```

**Algorithm 3:** Align CFG

```
Procedure AlignCFG(g: CFG, n: Node)
    subgraph $\leftarrow$ ExtractSubGraph(g, v)
    pathLengths $\leftarrow$ ComputeDistanceFromNode(subgraph, v)
    levels $\leftarrow$ { l | $u \in$ Nodes(subgraph) $\wedge$ pathLengths[u] = l }
    forall $l \in$ levels do
        levelNodes $\leftarrow$ { u | $u \in$ Nodes(subgraph) $\wedge$ pathLengths[u] = l }
        AlignNodes(subraph, levelNodes)
    end
Procedure AlignNodes(g: CFG, ns : NodeSet)
    index $\leftarrow$ 0
    while True do
        nodeLengths $\leftarrow$ { node: CountInstructions(node) | node $\in$
         Nodes(g) }
        candidates $\leftarrow$ {n | $n \in$ Nodes(g) $\wedge$ nodeLengths[n] =
         Max(nodeLengths) }
        referenceNode $\leftarrow$ SelectReferenceNode(candidates)
        referenceInstruction $\leftarrow$ GetNodeInstruction(referenceNode, index)
        forall node $\in$ {n | $n \in$ Nodes(g) $\wedge$ $n \neq$ referenceNode } do
            if index < nodeLength[node] $\wedge$
             Latency(GetNodeInstruction(node, index)) =
             Latency(referenceInstruction) then
                continue
            if IsBranch(referenceInstruction) then
                newInstruction $\leftarrow$ GetBranchInstruction()
                insertInstruction(node, newInstruction)
            else
                reg $\leftarrow$ SelectRegister()
                newInstruction $\leftarrow$
                 GetNOPInstruction(Latency(referenceInstruction), reg)
                insertInstruction(node, newInstruction)
        end
    end
Function SelectReferenceNode(candidates: NodeSet, index: Integer)
    for $n \in$ candidates do
        candidateInstruction $\leftarrow$ GetNodeInstruction(n, index)
        if $\neg$ (IsBranch(candidateInstruction) $\vee$
         IsReturn(candidateInstruction)) then
            return n
    end
    return candidates[0]
```

# Chapter 3

# Implementation

The algorithm has been implemented in X lines of Python code as part of the RetroWrite framework. RetroWrite is a binary rewriting tool developed for statically instrumenting programs. The authors are able to leverage relocation information present in position independent code to produce assembly files that can be reassembled into binaries. On top of this the framework provides a rewriting API that allows for flexible and expressive transformations of the binary code [3].

Implementing the algorithm on top of the RetroWrite frameworks allows for the alignment of instructions in existing binaries. This means that you do not need access to the source code ... **benefits of binary rewriting here**

The RetroWrite framework imposes some restrictions on the binary. The binary

- must be compiled as position independent code
- must be *x86_64*
- must contain symbols and cannot be stripped

[5]

The detection of secret dependent branches is not part of the algorithm or the implementation. The user has to provide the algorithm with the address of the target instruction. At the time of writing secret dependent branching instructions need to be identified through manual inspection. However, research has shown that static detection of these side channels is possible, though this is currently limited to the MSP430 architecture [7].

Intel provides some data regarding the latencies of commonly used instructions [1] but this data is not complete. To obtain better data Abel et. al developed novel algorithms to infer the latency throughput, and port usage based on automatically-generated microbenchmarks [2]. The authors claim that their results are more accurate and precise than existing work. Another source of data on instruction latencies is provided by Agner Fog who provides the results of his own measurements [4].

The data provided by Abel et. al is used as the primary source of instruction latencies. In the case where an instruction is not covered by their work the data provided by Agner Fog and Intel are used as a secondary source. If a program contains an instruction that is not covered by any of the datasets then the program cannot be aligned. The exception to this rule are branching instructions. There is no latency information available about these instructions in any of the sources. To account for this all branching instructions are aligned with new branching instructions. To preserve the control flow of the program the target of the branching instruction is equal to the address of the next instruction.

# Chapter 4

# Evaluation

## 4.1 Benchmark Suite

Winderix et. al. have created the first benchmark suite of programs with timing side-channel vulnerabilities. This suite consists of a collection of synthetic programs with a wide range of control-flow patterns as well as third party benchmark programs from different sources [9]. To evaluate the proposed algorithm a subset of this benchmark suite was selected.

All programs that contain loops inside vulnerable branches were discarded from the synthetic programs in the benchmark suite, since they are not supported by the proposed algorithm. The original authors of the Nemesis attack provide two case studies to demonstrate their attack. The first case study is a password comparison routine from the Texas Instruments MSP430 Bootstrap Loader (BSL). The second case study is secure keypad application that guarantees secrecy of its PIN code [8]. Both of these are included in the benchmark suite created by Winderix et. al and are also selected as a benchmark for the proposed algorithm.

The implementation of Nemesis and the benchmark suite created by Winderix et. al are implemented for the Sancus environment. Any pieces of code specific to this environment have been removed from the benchmark programs. The semantics of the programs remain unchanged.

One additional synthetic programs was added to the benchmark suite to evaluate a case that was not yet covered. This program contains a call to a function that modifies a non-local variable though a pointer. This function is only called in one branch of a secret-dependent branch.

## 4.2 Experiment Setup

The algorithm is evaluated using three metrics. The first metric aims to measure the effectiveness of the algorithm. A static analysis tool was developed to verify for a given program whether or not the program satisfies the Nemesis-sensitive property as specified in section 2.2. Given a program and a set of secret-dependent branches, this tool partitions instructions into sets according to their positions in secret dependent branches. Following the notation of section 2.2, let $ep$ be a secret dependent branch, and let $ep^n$ be the n'th instruction in a region, then define the set

$$ep_i = \{ep^n | i = n \wedge (ep^n \in region_{then}(ep) \vee ep^n \in region_{else}(ep)\} \tag{4.1}$$

The static analysis verifies that both the regions have the same number of execution points, and that for each set $ep_i$ it holds that all instruction have the same latency.

The second metric aims to measure the correctness of the algorithm. The algorithm is considered to work correctly if it does not change the program output. For each program in the benchmark suite a number of input values were determined such that all possible paths of the program control flow were covered. These values were supplied as inputs to both the original program and the balanced program, generating two output values. The output values were then compared to verify that the algorithm correctly modified the program without changing the output.

The effect on the program's performance is evaluated by measuring the increase in the sum of the latencies along paths in the programs CFG. To measure this increase CFG are constructed from the original binary and from the modified binary. For each path in the original CFG its corresponding path in the modified CFG is determined. TO do so a mapping is created that maps all nodes in the original CFG to their corresponding node in the balanced CFG. This mapping takes into account the condition of a branching instruction, and can be defined inductively. The root of the original CFG is mapped to the root of the root of the balanced CFG. If two nodes are mapped and they both have one successor then their successors are mapped. If two nodes are mapped and they have two successors, then then nodes that are reached if the branching condition is true are mapped, and those that are reached if the condition is false are mapped.

Formally, let $G$ denote the original CFG, and let $G'$ denote the modified CFG. Let $succ(n)$ be the successors of node $n$, and let $succT(n)$ be the successor of node N when the branching condition is true. Let $F$ be the function that maps between the two CFGs.

1. $F(root(G)) = root(G')$
2. $F(n) = n' \wedge succ(n) = \{s\} \wedge succ(n') = \{s'\}$
   $\implies F(s) = s'$
3. $F(n) = n' \wedge succ(n) = \{s, t\} \wedge succ(n') = \{s', t'\} \wedge succ_T(n) = s \wedge succ_T(n') = s'$
   $\implies F(s) = s', F(t) = t'$

Let $p$ be a path in $G$

$$p : p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_n$$

Then its corresonding path in $G'$ is defined as follows

$$p' : F(p_1) \rightarrow F(p_2) \rightarrow ... \rightarrow F(p_n)$$

This definition requires that $G$ and $G'$ are isomorphic. If during the first stage of the algorithm additional nodes were inserted in the CFG then this will not be true.

| Name | Effectiveness | Correct | Performance | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | path1 | path2 | path3 | path4 | path5 | path6 |
| call | Y | Y | 1.32 | 1.17 | | | | |
| call2 | Y | N | 1.25 | 1.22 | | | | |
| diamond | Y | Y | 1.35 | 1.06 | 1.06 | | | |
| fork | Y | Y | 1.43 | 1.15 | | | | |
| ifcompound | Y | Y | 1.31 | 1.26 | 1.11 | 1.11 | 1.09 | 1.09 |
| indirect | Y | Y | 1.44 | 1.32 | 1.20 | 1.11 | | |
| multifork | Y | Y | 1.80 | 1.58 | 1.41 | 1.41 | | |
| triangle | Y | Y | 1.30 | 1.16 | | | | |
| bsl | Y | Y | 1.42 | 1.00 | | | | |
| keypad | Y | Y | 1.67 | 1.55 | 1.45 | 1.02 | 1.12 | |

Figure 4.1: experiment results. Increase in performance is expressed as a percentage increase of the sum of the latencies along the path

Therefore before being able to evaluate the effect on runtime the first stage of the algorithm has to be reapplied on $G$ such that it is isomorphic to $G'$

To evaluate the effect on runtime performance the sum of the latencies along all relevants paths in G are compared to the sum of the latencies of their corresponding paths. A relevant path is a path that starts in secret-depdendent node and ends in a final node of one of the branches. Any nodes that do not belong to such a path are not affected by the algorithm and are therefore not considered in this evaluation.

## 4.3 Results

The results of the experiments are summarized in figure 4.1. The results show that the algorithm was able to ensure the Nemesis sensitive property holds for all programs, as verified by the static analysis tool described in the previous section.

In all but one test case the algorithm had no effect on the program output. The erroneous test case contains a call to a function that modifies the global state of the program in one of its secret dependent branches. During balancing of the program this function call is copied to the other branch. Because the function call has side effects the final output of the program is different.

The effect on performance ...

# Bibliography

[1] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[2] A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[3] S. Dinesh, N. Burow, D. Xu, and M. Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[4] A. Fog. Instruction tables:lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd, and via cpus, Mar 2021.

[5] HexHive. Hexhive/retrowrite.

[6] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979.

[7] S. Pouyanrad, J. T. Mühlberg, and W. Joosen. Scfmsp: Static detection of side channels in msp430 programs. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[8] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 178–195, New York, NY, USA, 2018. Association for Computing Machinery.

[9] H. Winderix, J. T. Mühlberg, and f. Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. 2021.