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. "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.

1.3 Introduction

This section outlines the approach to mitigate Nemesis style attacks. The cause of the vulnerability exposed by Nemesis-style attacks are differences in latency traces. Attackers are able to look at the differences in these latency traces and infer which secret-dependent branches were taken by the program, leaking information from the program. The algorithm outlined in this section aims to prevent this by aligning the latency traces along various paths by inserting additional instructions into corresponding nodes of two different branches, ensuring that there are no differences that can leak information.

The proposed algorithm performs a number of operations on a programs Control Flow Graph (CFG). The two main operations are the insertion of additional nodes into the graph and the alignment of a set of nodes. Because not all CFG structures are suitable for alignment the first stage of the algorithm inserts nodes into the graph to ensure alignment is possible. The second stage consists of alignment corresponding nodes in the graph.

Section 1.4 will define the property that needs to hold for a program in order for Nemesis-style attacks to be mitigated. Section 1.5 introduces the CFG data structure and translates the aforementioned property to such CFG structures. Finally, sections 1.6 and 1.7 describe the insertion and alignment of nodes, respectively.

1.4 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:

$$\begin{aligned} \forall ep^i \in region^{then}(ep) : \forall ep^j \in region^{else}(ep) \; such \; that \; i = j : \\ (s_{ep^i} \xrightarrow{t} s_{ep^i_{next}}) \wedge (s_{ep^j} \xrightarrow{t'} s_{ep^j_{next}}) \iff t = t' \end{aligned} \tag{1.1}$$

[1] 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.

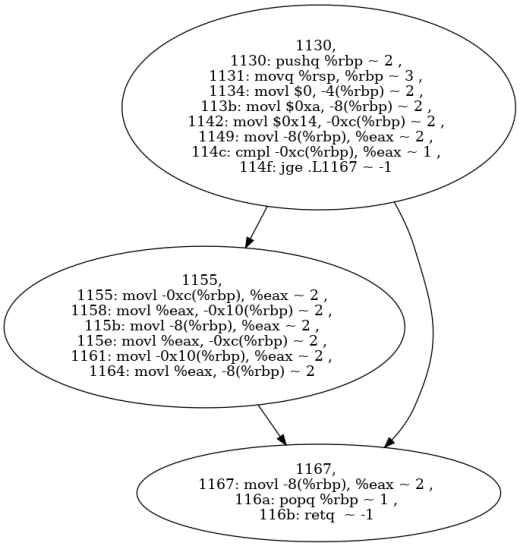
If this nemesis-sensitive property holds for a program then an attacker is not able to infer which branch was taken by the program based on instruction latencies.

1.5 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

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

(a) C program



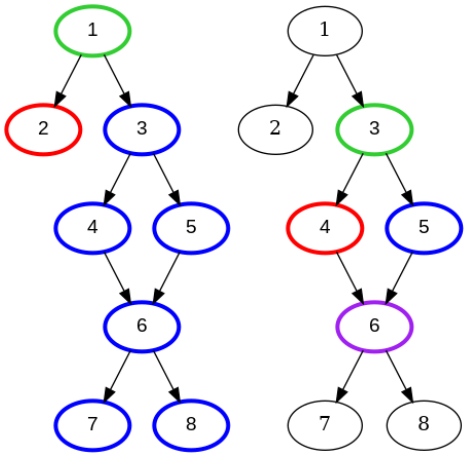
(b) Corresponding CFG

Figure 1.1: Example program with corresponding CFG

a contiguous sequence of instructions. Any branching instruction can only occur at the end of such a sequence, and an instruction that is the target of 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 1.1 shows an example of a such a CFG, along with the original program it is created from. The 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 1.1, 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 latencies between two nodes that are descendants of v_f cannot be used to infer information about the secret dependent node v. All nodes below v_f therefore 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.



(a) (b)

Figure 1.3: then-else regions for secret-dependent nodes

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

$$\forall n^i \in region^{then}(v) : \forall n^j \in region^{else}(v) \; such \; that \; i = j : \\ latencies(n^i) = latencies(n^j) \tag{1.2}$$

where $latencies(n)$ is a function mapping a node n to its latency sequence.

Figure 1.3 illustrates how the borders of each region is 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 1.2a 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 1.2b 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.

1.6 Equalising

There are 2 structures commonly 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, shown in figure 1.5. Before aligning the nodes an equalising step is applied to ensure these structures are do not occur in the CFG.

1.6.1 Problematic structure

The first such structure occurs when the program contains some sequence of instructions that is only executed if some condition is true. In this structure there will be some node that has two paths to it of different lengths. One path will contain the node that corresponds to the conditional instructions, while the other path will not contain this node. Because the paths have some overlapping nodes and because the paths have different lengths it is impossible to equalize their latency traces by inserting additional instructions, since inserting instructions in one of the paths will also insert instructions into the other path. If these paths start at a secret-dependent node it is therefore 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.

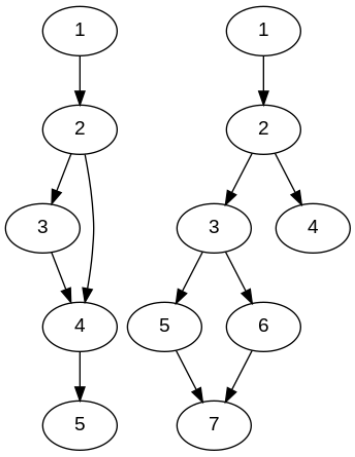
The nemesis-sensitive property as defined in section 1.2 entails that it is impossible for a node to satisfy the property if one of these structures occurs in its branches, since in both cases the regions have different depths. The first stage of the algorithm therefore consists of first equalizing all path lengths and then equalizing branches. Algorithms 1 and 2 depict pseudo-code for equalizing paths lengths and equalizing branches respectively.

1.6.2 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. This is done by applying a modified breadth-first search that stops early when the current node dominates all of the leaves reachable from n. By definition all paths between n and a leaf will pass through this node, in which case the any of the following nodes no dot have to be considered. Next the length of the longest path is computed from v to all nodes in the sub-graph. If there exists edges such that the length of the longest path to the tail and the length of the longest path to the head differ by more than one then there are at least 2 paths to the head of different lengths. In this case additional nodes are inserted between the tail and the head to equalize these path lengths.

1.6.3 Equalize branches

The branches of a CFG can be equalized in a similar way. Given some secret-dependent node v a subset of the graph's nodes are extracted. Next the lengths of the longest paths are computed for all leaves of the sub-graph, as well as the length of the longest path from the root to some leaf. If for some leaf the longest path to it is too short then additional nodes are inserted as predecessors to this leaf.



(a) optional node (b) unequal branches

Figure 1.5: problematic structures in CFG

1.7 Alignment

During the alignment stage the nodes of the CFG are aligned in a level-wise manner. During this stage additional instructions are inserted into nodes to ensure that corresponding instructions have the same latency. Algorithm 3 depicts pseudocode for this stage of the algorithm.

The level of a node is defined as being the distance between the root of the graph and the node. After the first stage of the algorithm all paths to a given nodes have the same length. This makes the level of a node a well defined value. The alignment stage iterates over all the levels of the sub-graph and aligns the nodes found at that level.

1.7.1 Basic Operation

The core of the alignment step consists of repeatedly selecting a reference instruction from one of the nodes and inserting instructions into the other nodes such that the latencies match. The node from which the reference instruction is selected is called the reference node. The reference node can change throughout the algorithm.

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. Let n_{max} be the number of instructions in the longest node. The set of candidate nodes then consists of all nodes that have n_{max} instructions. The reference node is then selected from this set of candidates.

An index variable is used to keep track of the position of the reference instruction. This variable is initially zero and is incremented every iteration. Any instructions

that have an index smaller than this variable are balanced. The reference instruction is selected by first selecting the reference node and then selecting the instruction in this node that has an index equal to the index variable.

Given a reference instruction, the algorithm iterates over all nodes that are not the reference node and verifies if the corresponding instruction has the same latency. If the two latencies are not equal, or if the node has no corresponding instruction, then a new instruction is inserted into the node at the index equal to the index variable. Once this has been done for all nodes then all instructions with an index smaller than or equal to the index variable will be balanced and the index variable can be incremented.

1.7.2 Selecting the Reference Node

If there are multiple candidates nodes then the algorithm selects one of the nodes where the corresponding reference instruction is not a branching instruction or a return statement. From this set a reference node is arbitrarily selected. If there are no such nodes then any node can be selected as the reference node.

1.7.3 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.

The function is statically analyzed to determine which registers are free to use for this purpose. There are two types of free registers. A register can be free because its current value is no longer used, i.e. because it 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.

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, and 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.

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.

1.7.4 branching instructions

In the cases where the reference instruction is a branching instruction, the newly inserted instruction will also be a branching instruction. The target of the branching instruction is the successor of the node where the node is inserted.

Algorithm 1: Equalize Path Lengths

```
1  Procedure EqualizePathLengths(g: CFG, v: Node)
2      subgraph ← ExtractSubGraph (g, v)
3      longestPathLengths ← ComputeLongestPathLengths(subgraph, v)
4      forall (u, v) ∈ Edges(subgraph) do
5          diff ← longestPathLengths[u] - longestPathLengths[v]
6          if diff > 1 then
7              head ← v
8              for i ∈ 1, 2, ..., diff-1 do
9                  newNode ← CreateNode()
10                 InsertNodeBetween(newNode, u, head)
11                 head ← newNode
12             end
13     end
14 Function ComputeLongestPathLengths(g: CFG, s: Node)
15     dist = {n : -1 | n ∈ Nodes(g) }
16     dist[s] ← 0
17     forall n ∈ TopologicalOrder(g) do
18         forall succ ∈ Successors(n) do
19             dist[succ] ← Max(dist[succ], dist[n] + 1)
20         end
21     end
22     return dist
23 Function ExtractSubGraph(g: CFG, n: Node)
24     immedidateDominators ← computeImmediateDominators(g, n)
25     leafDominators ← { u ∈ Nodes(g) |(u, v) ∈ leafDominators ∧v ∈
        Leaves(g) }
26     if | leafDominators | = 1 then
27         dominator ← leafDominators[0]
28     else
29         dominator ← ∅
30     subgraphNodes ← [n]
31     adjacentNodes ← Successors(n)
32     while | adjacentNodes | > 0 do
33         currentNode ← adjacentNodes[0]
34         adjacentNodes ← adacentNodes \ {currentNode}
35         if currentNode = dominator then
36             return subgraphNodes
37         adjacentNodes ← adjacentNodes ⋃
            (Successors(currentNode) \ adjacentNodes)
38     end
39     return subgraphNodes
```

Algorithm 2: Equalize Branches

```
1  Procedure EqualizeBranches(g: CFG, n: Node)
2      subgraph ← ExtractSubGraph (g, v)
3      longestPathLengths ← ComputeLongestPathLengths(subgraph, v)
4      maxPathLength ←
         Max({longestPathLengths[v] | v ∈ Leaves(subgraph)})
5      forall leaf ∈ Leaves(subgraph) do
6          diff ← longestPathLengths[leaf] - maxPathLength
7          if diff > 0 then
8              v ← leaf
9              for i ∈ 1, 2, ..., diff do
10                 newNode ← CreateNode()
11                 AddNode(g, newNode)
12                 forall p ∈ Predecessors(v) do
13                     AddEdge(g, (p, newNode))
14                     RemoveEdge(g, (p, v))
15                 end
16                 AddEdge(g, (newNode, v))
17             end
18     end
```

Algorithm 3: Align CFG

```
1  Procedure AlignCFG(g: CFG, n: Node)
2      subgraph ← ExtractSubGraph(g, v)
3      pathLengths ← ComputeDistanceFromNode(subgraph, v)
4      levels ← { l | u ∈ Nodes(subgraph) ∧ pathLengths[u] = l }
5      forall l ∈ levels do
6          levelNodes ← { u | u ∈ Nodes(subgraph) ∧ pathLengths[u] = l }
7          AlignNodes(subraph, levelNodes)
8      end
9  Procedure AlignNodes(g: CFG, ns : NodeSet)
10     index ← 0
11     while True do
12         nodeLengths ← { node: CountInstructions(node) | node ∈
            Nodes(g) }
13         candidates ← {n | n ∈ Nodes(g) ∧ nodeLengths[n] =
            Max(nodeLengths) }
14         referenceNode ← SelectReferenceNode(candidates)
15         referenceInstruction ← GetNodeInstruction(referenceNode, index)
16         forall node ∈ {n | n ∈ Nodes(g) ∧ n ≠ referenceNode } do
17             if index < nodeLength[node] ∧
                Latency(GetNodeInstruction(node, index)) =
                Latency(referenceInstruction) then
18                 continue
19             if IsBranch(referenceInstruction) then
20                 newInstruction ← GetBranchInstruction()
21                 insertInstruction(node, newInstruction)
22             else
23                 reg ← SelectRegister()
24                 newInstruction ←
                    GetNOPInstruction(Latency(referenceInstruction), reg)
25                 insertInstruction(node, newInstruction)
26         end
27     end
28 Function SelectReferenceNode(candidates: NodeSet, index: Integer)
29     for n ∈ candidates do
30         candidateInstruction ← GetNodeInstruction(n, index)
31         if ¬ (IsBranch(candidateInstruction) ∨
            IsReturn(candidateInstruction)) then
32             return n
33     end
34     return candidates[0]
```

Chapter 2

Implementation

Chapter 3

Evaluation

3.1 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 [3]. 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 [2]. 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.

3.2 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 1.4. 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 1.4, let ep be a secret dependent branch, and let ep^n be the n'th instruction in a region, then define the set

$$ep_i = \{ep^n | i = n \wedge (ep^n \in region_{then}(ep) \vee ep^n \in region_{else}(ep)\} \tag{3.1}$$

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 corresponding paths in the control flow of the original program and the balanced program. To this end the CFGs are constructed from the original binary as well as from the balanced binary. To determine for a path in the original CFG what its corresponding path is in the modified CFG 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

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

Then its corresonding path in G' is defined as follows

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

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

Name	Effectiveness	Correct	Performance					
			path1	path2	path3	path4	path5	path6
bsl	Y	Y	1.42	1.00				
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		
keypad	Y	Y						
multifork	Y	Y	1.80	1.58	1.41	1.41		
triangle	Y	Y	1.30	1.16				

Figure 3.1: experiment results. Increase in performance is expressed as a percentage increase of the sum of the latencies along the path

true. 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.

3.3 Results

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] 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.

[2] 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.

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