KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

NemesisGuard: Mitigating Interrupt Latency Side Channel Attacks With Static Binary Rewriting

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

Gilles De Borger

Contents

Abstract

Samenvatting

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Protected Module Architectures

Because of the increasing popularity of IoT devices more and more embedded computing devices are being connected to the Internet. These devices are often more susceptible to being exploited because they support software extensibility. Additionally because these devices are connected to a network the risk increases since attacks can be done remotely. An important technique for securing such devices is the use of hardware support for virtual memory and processor privilege levels. The OS can build on this support to isolate a process from any other malicious processes on the device. However, this introduces a sizable software layer however which is difficult to get sufficiently secure [23]. If the attacker controls the OS then its capabilities for attacking a process on the devices increase considerably.

Maene et al. state that *the goal of trusted computing is to develop technologies which give users guarantees about the behaviour of the software running on their devices.* An important aspect of trusted computing is therefore to protect software even when attackers have full control of the system. One means of achieving this is through the use of Protected Module Architectures (PMAs). These architectures seperate critical components into protected modules, also called enclaves, that are isolated from one another through hardware [16].

A number of Protected Module Architectures (PMAs) have been developed to address this problem, both by researchers and industry. PMAs have been developed for both low-end microcontrollers found in embedded systems [14, 9] as well as high-end processors [10]. One architecture developed for embedded systems is Sancus. Sancus is a security architecture that can provide strong isolation guarantees on networked embedded systems, and has been implemented on a modified TI MSP430 micro-controller [19]. At the higher end of the spectrum there is Intel SGX. Intel SGX is an extension added to the Intel architecture that allows applications to instantiate enclaves. Enclaves are areas in the application's memory that are protected from access from outside of the enclave, even from privileged software such as the OS [17].

Research has shown that it is still possible to extract information from protected applications in PMAs. In their work Xu et al. [27] introduce a novel type of side-

channel attack called controlled-attacks. These attacks are categorized by untrusted operating systems that create side-channels through its extensive control of the system. The authors were able to leverage the OS's high degree of control over the system to extract large amounts of data from applications which were until then safe from side-channel attacks.

1.2 Nemesis

More recently Van Bulck et al. [24] developed Nemesis, a controlled-channel attack that leverages the interrupt mechanism to extract sensitive information from enclaved applications. The authors were able to exploit timing differences in the latency between the arrival of an interrupt request (IRQ) and the execution of the first instruction in the interrupt service routine (ISR). They state that their attack is *based on the key observation that an IRQ during a multi-cycle instruction increases the interrupt latency with the number of cycles left to execute.* By carefully and deliberately interrupting a process at the right time the authors were able to infer the duration of the interrupted instruction. Potential attackers can use this information to determine where the instruction is situated in the program's control flow. When the instruction is part of a secret-dependent branch the attacker is able to infer some information about the secret, successfully leaking sensitive information from the program. Van Bulck et al. [24] showed that this attack is applicable to the whole computing spectrum. They were able to apply their attack to the aforementioned Sancus architecture as well as Intel SGX enclaves.

Figure 1.1 illustrates how such an attack might work with a piece of assembly pseudocode. An attacker who is in control of the OS could carefully interrupt the program right after the conditional jump at line 5. Depending on the value of register r1 the next interrupted instruction is either the addition instruction at line 4 or the multiplication instruction at line 7. By measuring the interrupt latency the attacker can infer which of the two instructions was being executed at the time of the interrupt and, more importantly, infer if the value in register r0 is equal to 0.

1.3 Binary Rewriting

Binary rewriting is the alteration of a compiled program without having the source code at hand. Applications of binary rewriting include observing programs during execution, optimizing programs using run-time patching, and hardening applications against attacks. In the case of dynamic binary rewriting the rewriting happens during execution of the program. Static binary rewriting, on the other hand, occurs before the binary is executed [25]. Binary rewriting tools have been developed for both low-end architectures found in embedded devices [21] as well as high-end architectures found in home computers and servers [8, 6, 7].

```
1       CMP r1, $0
2       JEQ .l1
3       .l1:
4       ADD r1, r2              ; 1 cycle instruction
5       JMP .end
6       .l2:
7       MUL r1, r2              ; 2 cycle instruction
8       JMP .end
```

Figure 1.1: Assembly pseudo-code with a secret-dependent branch that is vulnerable to Nemesis attack

1.4 Related Work

Software-based approaches

A large number of countermeasures have been proposed for closing timing side-channels. These can broadly be categorized as being either software-based or hardware-based. Hardware-based solution are based on modification to the architecture, whereas software-based solutions are implemented at the language level [3].

Popular software-based approaches to closing timing-leaks include constant-time policies and the program counter model [3]. Constant time policies require that memory access and control-flow should not depend on secret data. Unfortunately writing code that adheres to these policies can be difficult since it requires knowledge of the compiler and requires developers to deviate from conventional programming practices. A number of solutions have been proposed to verify if a program adheres to constant-time policies [4, 3].

Molnar et al. [18] first introduced the program counter model in their work, proposing methods for the detection and mitigation of control-flow side channel attacks. The authors consider the case where an adversary is able to make an observation of a side channel at each step of the computation. The result is a sequence of observations $T = (T_1, T_2, ...T_n)$ called a *transcript*. The *program counter model* or *PC model* is then the model where each value of the transcript is the processor's program counter during the computation. A program is then said to be PC-secure if the program if this transcript is secure. The authors state that *any program that is PC-secure will also be secure against timing attacks*. Based on this definition of PC-security the authors introduce a code transformation for creating PC-secure C code. The authors note that they made a number of assumptions in their work. Although these assumptions do not hold for a number of architectures the authors are confident that they are reasonable for some embedded devices.

Hardware-based approaches

An orthogonal approach is to close timing leaks using hardware-based solutions. Recently Busi et al. [5] proposed an approach to extend architectures with non-interruptible enclaved execution such that they can also support interruptions without breaking the existing isolation properties. Based on this approach they proposed a design for interruptible enclaves that are resistent against interrupt attacks. This design is based on an earlier version of Sancus with non-interruptible enclaves. They modify the architecture to add padding cycles whenever the enclave is interrupted, effectively closing timing leaks.

A limitation of their approach is that their design is based on the assumption that the timing of instruction is predictable. This is generally not the case for more complex architectures such as Intel's x86_64. Additionally their approach requires hardware modifications which means it cannot be applied to off-the-shelf and existing devices.

1.5 Thesis Goal and Outline

This paper presents a novel algorithm for automatically transforming a program in order to remove any timing leaks. It achieves this by addressing the core cause of the vulnerability: differences in latencies between corresponding instructions in secret-dependent branches. Corresponding instructions are instructions that are the same distance away from a branching instruction. The proposed algorithm inserts additional instructions such that all corresponding instructions have the same latency without changing affecting the program output.

The main contributions of this paper are:

1. The paper presents a novel algorithm for automatically transforming a program to remove any timing leaks.
2. The paper presents an implementation of this algorithm for the Intel x86-64 architecture.
3. The paper presents an evaluation of the algorithm based on a suite of benchmark programs.

Chapter 2

Design

This chapter details the design of the proposed algorithm. Section ... **TODO**

2.1 Algorithm Goals

The root cause of the vulnerability exposed by Nemesis is a difference in the latencies of two corresponding instructions in different branches of a conditional jump. In this context two instructions correspond if they are the same distance away from the conditional jump. Van Bulck et. al [24] convincingly demonstrated that is is possible to exploit this vulnerability by inspecting latency traces of the program. To create a latency trace the attacker configures a timer such that each instruction of the program is interrupted, effectively single-stepping through the program. The attacker then implements the Interrupt Service Routine (ISR) such that it logs the latency of the interrupted instruction [13, 22]. By varying the input he creates a set of latency traces that will reflect any differences in latencies of corresponding instructions. Through careful inspection the attacker is able to infer information about which path the control flow of the program followed for a given input. If the corresponding instructions are part of a secret-dependent branch then the attacker is then able to infer some information about the value of the secret.

If the architecture has completely deterministic instruction latencies then a single latency trace per input value is sufficient for an attacker to leak information. In the presence of advanced architectural features, such as paging, caching, and out-of-order execution, instruction latencies are not deterministic. Van Bulck et. al [24] show that in these cases is still possible to leak information from the program by correlating latency traces from repeated executions over the same input. The authors demonstrate that the latency measurements follow a normal distribution across all executions and are able to use basic statistical measures to create an overall latency trace of the program.

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-dependent

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:

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

[20]

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 their latencies should be the same.

If a program satisfies the nemesis-sensitive property then an attacker will not be able to leak information from the program based on its latency traces. For simple architectures with deterministic instruction latencies the resulting latency traces will be indistinguishable. Architectures with more complex features will still have minor differences in the resulting traces due to the random nature of the instruction latencies. However these differences will not be large enough to be statistically significant. As a result the attacker will not be able to infer any information about the control flow of the program.

The goal of the proposed algorithm is to ensure that latency traces cannot leak information by enforcing the nemesis-sensitive property. The algorithm inserts additional instructions into secret-dependent branches such that corresponding instruction always have the same latency. These instruction are carefully constructed such that they have no effect on the program output.

2.2 CFG

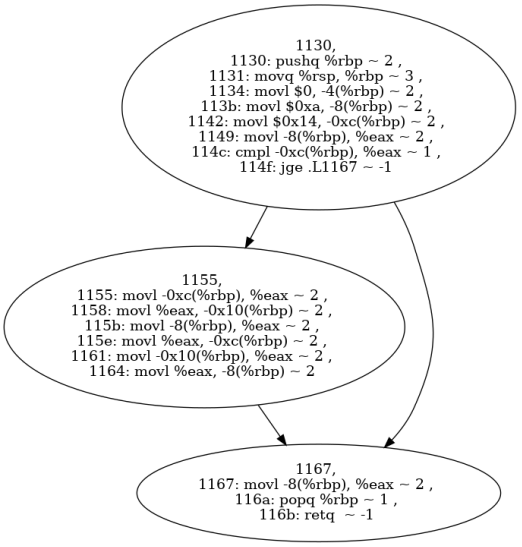
The algorithm makes modifications to a program's functions by manipulating the Control Flow Graph (CFG), a data structure that represents the control flow of a function. If a program has multiple functions then each function is represented by a separate CFG. A CFG consists of nodes V and directed edges E. Each node v contains a contiguous sequence of instructions that is always executed as a whole. This implies that a branching instruction can only be found at the end of a node, and an instruction that is the target of a branching instruction 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 two successors. 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. This latency sequence is 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

```
1  int main(){
2          int a = 10;
3          int b = 20;
4          if (a < b){
5                  int temp = b;
6                  b = a;
7                  a = temp;
8          }
9          return a;
10 }
```

(a) C program



(b) Corresponding CFG

Figure 2.1: Example program with corresponding CFG

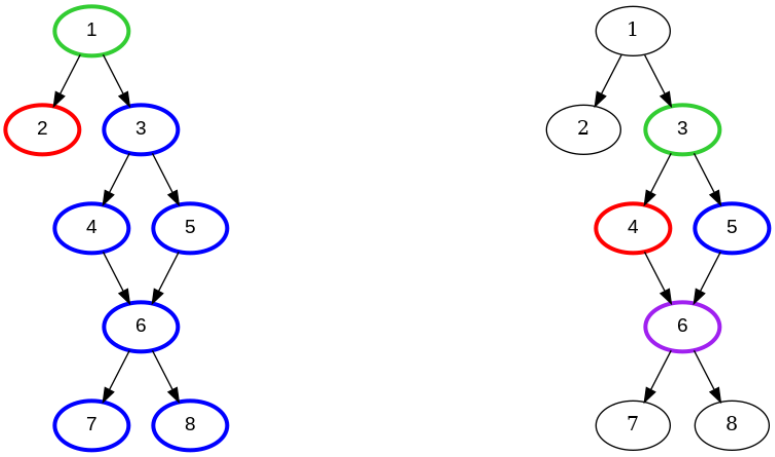
the path. Figure 2.1 shows an example of a such a CFG, along with the original program it is created from. The CFG also contains the latency for each instruction. Note that by convention the only node with no incoming edges is considered the starting node of the CFG.

Following the property described in property 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 consist of all nodes reachable from v through one of its outgoing edges.

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. Let $n^i \in region(v)$ be a node such that there is a path going to it from node v of length i. 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

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



(a) (b)

Figure 2.2: then-else regions for secret-dependent nodes

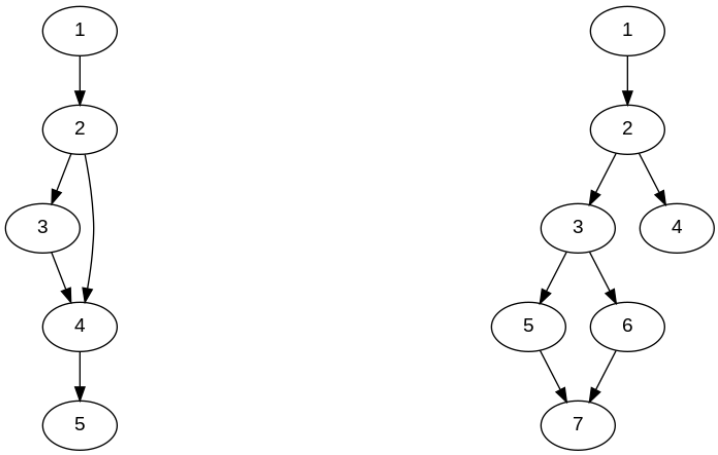
where $latencies(n)$ is a function mapping a node n to its latency sequence. This property states that the latency sequence of any two nodes that are the same distance away from some secret-dependent node must have identical latency sequences. If this property holds then the critical sections of latency traces will be identical and cannot be used to infer information about the secret-dependent branch.

Figure 2.2 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 no 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.

2.3 Equalising

There are two structures that can occur in a function's CFG 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 2.3. The first stage of the algorithm consists of inserting nodes such that these structures no longer occur.

The first such structure, illustrated in figure 2.3a, occurs when a function contains some sequence of instructions that is only executed if some condition is true. Analogously the corresponding CFG will contain a node that is only reached if the condition is true. A consequence of this is that there will be some node in the CFG that has at least two paths to it. One path will contain the conditional node while the other will not. Additionally, the path with the conditional node will be longer than the other path. Because the nodes in the shorter path form a subset of the



(a) optional node (b) unequal branches

Figure 2.3: problematic structures in CFG

nodes in the longer path it is impossible to modify the shorter path without also modifying the longer path, making it impossible to ever equalize the traces along the two paths.

The second problematic structure occurs when one of the branches is shorter than the other one, as shown in figure 2.3b. If one branch is shorter than the other then there will be some nodes in the longer branch that have no corresponding nodes in the shorter branch. This makes 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 inserting additional nodes into the CFG such that all path lengths to a given node are equal as well as all branch depths. Algorithms 1 and 2 depict the procedures for equalizing paths lengths and equalizing branches respectively.

2.3.1 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 2.2. The procedure *ExtractSub-graph*, shown in algorithm 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 [15]. 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 2.2.

2.3.2 Equalize Path Lengths

The procedure for equalizing all path lengths is show in algorithm 1. Let v be the secret-dependent node. First the subset of the graph is 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 exist 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 predecessor of v and has length $d(v)$.

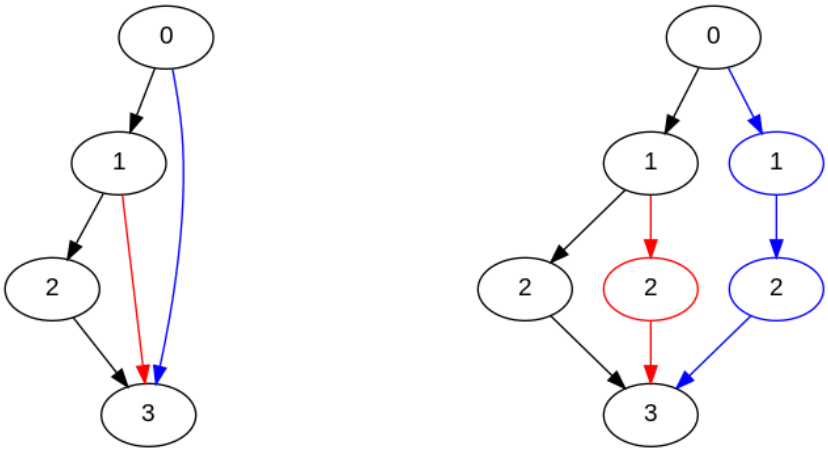
The solution is based on the observation that the edge (u, v) is a part of the shorter path. Additional nodes can be inserted between u and v to increase its length until it is as long as the longest path. All other paths to v will be unaffected.

The procedure iterates over each edge of the subgraph. If the distances to u and v differ by more than one then nodes are inserted into the edge between u and v until the path is sufficiently long. The number of nodes that need to be inserted is equal to $d(v) - d(u) - 1$

Figure 2.4 illustrates this procedure. Each node in the graph is labeled with the length of the longest path to it. Figure 2.4a has two edges drawn between nodes where the difference between the path lengths is more than one. These edges are marked in red and blue. Additional nodes are inserted to get the graph shown in figure 2.4b.

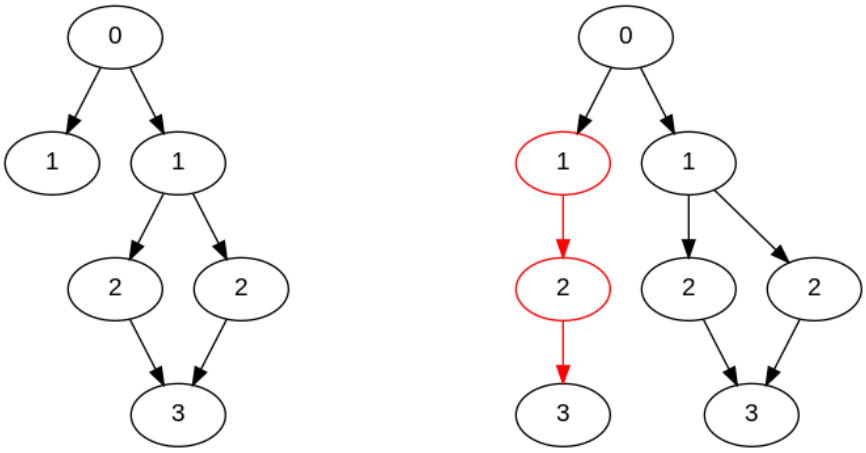
2.3.3 Equalize branches

The branches of the CFG can be equalized in a similar way. The procedure for doing so is shown in algorithm 2. Given some secret-dependent node v the subset of the CFG is extracted that contains $region_{then}(v)$ and $region_{else}(v)$. 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 one of the leaves of the CFG. The procedure 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. Any additional nodes have to be inserted as predecessors because the final instruction in a leaf is a return statement.



(a) original CFG (b) equalized CFG

Figure 2.4: Equalizing path lengths



(a) original CFG (b) equalized CFG

Figure 2.5: Equalizing branch depths

Figure 2.5 shows how nodes are inserted in order to equalize branch depths. The CFG in figure 2.5a has one branch with depth equal to one. Two additional nodes are added as ancestors to the branch's leaf in order to increase the branch depth to three. The result of this process is shown in 2.5b.

2.4 Alignment

During the second stage of the algorithm 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.

2.4.1 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. 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 dummy instruction is inserted at positioni_{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 at position i_{ref} have the same latency and the variable can be incremented.

2.4.2 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.

A branching instruction is a special case that will result in the insertion of dummy branching instructions into the other nodes. Additionally a branching instruction cannot be inserted into he middle of a node since this will change the program's control flow. A node can therefore not be selected as the reference node when the instruction at position i_{ref} is a branching instruction, unless during the very last iteration.

If among the set of candidate nodes there is at least one node with a non-branching instruction at position i_{ref} then this node is selected as the reference node. If there are multiple such nodes then a candidate node can be selected arbitrarily. The case where all candidate nodes have a branching instruction at position i_{ref} can only occur during the very last iteration. At this point any of the candidate nodes are suitable and one is selected arbitrarily again.

2.4.3 Constructing NOP instruction

Any instructions that is inserted into the program can have no effect on the program outcome. In this regard they are the same as the no-operation instruction and are referred to as NOP instructions. For each latency class a template NOP instruction has been determined. For some latency classes a NOP instruction exists that has no effect on the program state. These can be inserted into the program as-is. For other latency classes the NOP instruction modifies some register value. In these cases 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 in order to guarantee that the program outcome is not changed.

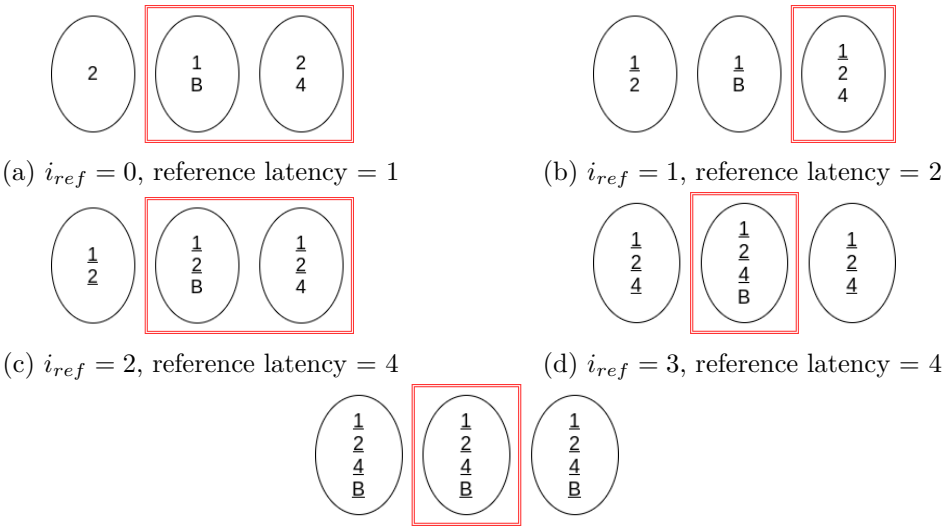
There are two types of free registers. Firstly a register can be free because its current value is no longer used. This occurs when the register is overwritten at some later point without being read first. Alternatively a register can be free because it isn't used anywhere in the current function. In 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 a push instruction is inserted to store the register value onto the stack. A pop instruction is then inserted in every leaf of the CFG to restore the register value. 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. A dummy label is inserted at the start of the node's successor. This label then becomes the target of the branching instruction.

If the reference instruction is a call to a function then the NOP instruction will be a call to the same function. The instruction is simply duplicated into the current node. If the function contains no secret dependent node then this ensures that any latency traces cannot leak information from the program. Otherwise the function that is called needs to be aligned as well. It is only safe to insert a call to a function this way if the function in question has no effects on the program state. If the function does modify the program state then inserting additional calls can result in erroneous program outputs.



(a) $i_{ref} = 0$, reference latency = 1

(b) $i_{ref} = 1$, reference latency = 2

(c) $i_{ref} = 2$, reference latency = 4

(d) $i_{ref} = 3$, reference latency = 4

(e) $i_{ref} = 4$, nodes aligned

Figure 2.6: Example of alignment of a set of three nodes

2.4.4 Example

Figure 2.6 illustrates the different iterations of the alignment of a set of three nodes. Each node is labeled with its latency sequence. The set of instructions that have been aligned are underlined. Each iteration the variable i_{ref} is incremented by one. The set of candidate nodes is marked in each iteration by a red rectangle. If there are multiple candidate nodes with non-branching instructions at position i_{ref} then the first node is arbitrarily selected as the reference node.

2.5 The algorithm

2.5.1 Cycles

The operations described in section 2.3 require the CFG to be acyclic. To account for this all cycles are removed from the CFG beforehand and later restored. The edge that needs to be removed is determined based on the depths of the nodes in the cycle. The depth of a node is defined as being the length of the longest path to the node from the root. Removing the cycle is done by removing the edge from that graph that connects the deepest node to the most shallow node. The tail and head are stored for all edges that are removed so that they can later be restored.

The removal of these edges has no effect on the operations of the algorithm if the cycle is not nested inside the branch of a secret dependent node. Cycles within a branch of a secret dependent node are not supported by the proposed algorithm. In this case the algorithm correctly aligns the nodes of the branches but the latency traces will still not be identical. The branch that contains the cycle will be executed a higher number of times, resulting in a longer latency trace.

2.5.2 Secret-dependent branches

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 [20].

2.5.3 Closing timing leaks

Algorithm 4 illustrates the top-level operation of the algorithm. The secret-dependent branching instructions are passed to the algorithm as arguments. Based on the given instructions the secret-dependent nodes of the CFG are determined. These are all nodes that contain a secret-dependent instruction. First the equalizing operations are applied for each secret-dependent node. Only once all necessary nodes have been inserted is the CFG aligned. This order of operations ensures that the algorithm works correctly even when secret-dependent node is nested inside the branch of another secret-dependent node. This can occur when, for example, a program contains a nested if-statement. Cycles are removed from the CFG and restored as described in the previous section.

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 ← { }
31     if domintor ≠ ∅ then
32         forall path ∈ computeAllPaths(g, n, dominator) do
33             subgraphNodes ← subgraphNodes ∪ ({p|p ∈
                path} \ subraphNodes)
34         end
35     else
36         forall l ∈ Leaves(g) do
37             forall path ∈ computeAllPaths(g, l, dominator) do
38                 subgraphNodes ← subgraphNodes ∪ ({p|p ∈
                    path} \ subraphNodes)
39             end
40         end
41     subgraphEdges ← {(u, v)|u ∈ subgraphNodes ∧ v ∈
16      subgraphNodes ∧ (u, v) ∈ Edges(g) }
42     return (subgraphNodes, subgraphEdges)
```

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, v: Node)
2      subgraph ← ExtractSubGraph(g, v)
       // Compute distances from v to all nodes in subgraph
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]
```

Algorithm 4: Close timing leaks

```
1  Procedure CloseTimingLeaks(g: CFG, instrs: [Instruction])
2      targetNodes ← {n|n ∈Nodes(g)∧∃instr ∈ instrs : instr ∈ n}
3      removedEdges ← RemoveCycles(g)
4      forall node ∈ targetNodes do
5          EqualizePathLengths(g, node)
6          EqualizeBranches(g, node)
7      end
8      forall node ∈ targetNodes do
9          AlignCFG(g, node)
10     end
11     RestoreCycles(g, removedEdges)
```

Chapter 3

Implementation

3.1 Binary Rewriting

The algorithm has been implemented as a binary rewriting tool. This approach has a number of advantages. Firstly the algorithm can be applied to existing binaries. It does not require recompilation of original code and does not require the source code of the program. This means that it is suitable for use on commercial off-the-shelf binaries deployed in the field. Additionally there is no need for a modified compiler of hardware.

3.2 RetroWrite

The algorithm is implemented for the Intel x86_64 architecture. It is written in X lines of Python code as part of the RetroWrite framework. RetroWrite is a binary rewriting tool developed for statically instrumenting C and C++ binaries. The authors are able to leverage relocation information present in position independent code to reconstruct assembly files from a compiled binary. These files can be modified and reassembled into binaries. To do so the framework provides a rewriting API that allows for flexible and expressive transformations of the reconstructed assembly code [7]

The RetroWrite frameworks implements a logical abstraction for rewriting passes to operate on. These come in the form of data structures that represent the logical units of a program. Each of these data structure provide an interfaces for analyzing and modifying the underlying data. One such logical abstraction is the *InstructionWrapper*. This datastructure stores, among other things, the instruction address, the mnemonic, and the operand string, and provides an interface for modifying the underlying instruction and for prepending or appending additional instructions. The *Function* datastructure contains a set of instructions and a function that maps each instruction to all instruction that can follow it [7, 12].

The proposed algorithm is implemented as an additional abstraction layer on top of these data structures. Each node of the CFG consists of a sequence of *InstructionWrapper*s. The edges of the CFG are reconstructed based on the instruction

mapping stored in the *Function* instances. The CFG data structure implements an interface for the insertion of additional nodes into the graph and the insertion of additional instructions into nodes. This interface is built on top of the RetroWrite API, so all modifications to the CFG result in modification to the underlying *InstructionWrapper* instances. Once all necessary modifications have been made to the CFG the instructions are written to an assembly file using functionality provided by RetroWrite. This file can be then be compiled using any off-the-shelf compiler to create an executable.

The RetroWrite framework imposes some restrictions on the binary. The binary must be compiled as position independent code, it must contain instructions from x86_64 architecture, and it cannot be stripped of symbols [12]. As a result the implementation only supports binaries that meet these restrictions.

3.3 Instruction Latencies

The construction of NOP instructions as described in section 2.4.3 is based on data that measures the latency of instructions in the x86-64 architecture. 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. [2] developed novel algorithms to infer the latency throughput and port usage based on automatically-generated microbenchmarks. 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 [11].

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 it 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. [26] 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. 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 [24]. 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 whether or not program satisfies the Nemesis-sensitive property as specified in section 2.1. 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 2.1, 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 CFGs 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

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

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 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.

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. A check mark in the second column indicates that all timing-leaks were closes. A check mark in the third column indicates that the program output has not changed. The third column contains the increase in the sum of the latency along various paths in the program, expressed as a percentage increase.

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. This effectively closes the timing leaks found in the program.

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 performances ranges from an increase by a factor of 1.8 to no increase at all. There are two factors that cause larger increases in the sum of the latencies along a path. The first is the difference in depths between two secret dependent branches. If one branch is significantly deeper then a larger number of nodes will be inserted in the other branch, lengthening the paths found in the shorter branch. Because each inserted node has to be aligned, a larger number of instructions will be inserted.

This effect is the cause for the relatively large increase in latencies in the benchmark program *multifork*. This program contains a secret-dependent switch statement. This is equivalent to three nested secret-dependent if statements. The path followed by the control flow if the very first case matches the expression is much shorter than the path followed if the very last expression matches. Because all paths need to have the same length the number of instructions inserted in this shorter path is much larger.

The second factor is the presence of a larger conditional code block. All instructions found in the conditional node also have to be inserted in the path where the condition is false. As the number of instruction in the conditional node increases so does the sum of the latencies in the other path.

This is reflected in the difference between the experiment results for the benchmarks *triangle* and *fork*. The structure of the CFGs is similar for both these benchmarks as both contain a conditional node. Benchmark *fork* assign the result of

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

Figure 4.1: Results of the evaluation experiments

an expression to a variable inside this, whereas *triangle* simply assigns a constant. The number of instruction in the optional node is therefore larger in *fork*, resulting in a larger effect on performance in *fork*.

Chapter 5

Conclusion

5.1 Discussion

One limitation of the proposed algorithm is the lack of support for cycles within secret-dependent branches. However, the evaluation shows that the algorithm is effective in closing timing leaks in programs where this does not occur. This indicates that it is possible modify the algorithm such that it is able to close these leaks even in the presence of cycles.

The root cause of the issue is the fact that a section of the first branch will be executed a higher number of times than the corresponding section in the second branch. As a result one of the latency traces will be longer, even when the relevant nodes are aligned. A solution that addresses this would have to make more extensive changes to the program. Before aligning the nodes, the structure of the cycle would have to be duplicated into the second branch such that the corresponding section is is executed the same number of times. This requires additional analysis to determine which register contains the loop counter and how many time it is incremented. The duplication of the cycle also requires more significant changes than those implemented by the current algorithm. Due to the added complexity such a solution is not included in the proposed algorithm and is left to future research.

A second limitation is the incomplete coverage of the latency data. There are certain instructions for which there is no available data. If these instructions are encountered in branches of a secret-dependent branching instruction then the algorithm is not able to close the timing leaks. However this issue was not encountered during evaluation of the algorithm. This indicates that the most commonly used instructions are present in the data and that the coverage of the data is sufficient to close timing leaks in most programs.

The evaluation of the effectiveness of the algorithm is based on a statistical analysis of the program before and after alignment. If the instruction latencies are fully deterministic then the static analysis tool can correctly predict the actual run-time instruction latencies. The resulting analysis is then sufficient for demonstrating that the algorithm correctly closes all timing leaks. In the presence of advanced micro-architectural features the instruction latencies are to some extend random,

however. As a result the run-time latencies diverge from the predicted latencies. Although the results indicate that all timing leaks are closed it is possible that some differences still exist between branches because of these random variations. Because of this additional experiments are needed to fully verify if the algorithm is effective for complex architectures with non-deterministic latencies. Empirical measurements in the form of latency traces are needed to determine if an attacker can still distinguish between branches of a secret-dependent conditional node.

5.2 Future work

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] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[4] S. Blazy, D. Pichardie, and A. Trieu. Verifying Constant-Time Implementations by Abstract Interpretation. In *European Symposium on Research in Computer Security*, 22nd European Symposium on Research in Computer Security, Oslo, Norway, Sept. 2017.

[5] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors: Extended version, 01 2020.

[6] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton. Instruction punning: Lightweight instrumentation for x86-64. *SIGPLAN Not.*, 52(6):320–332, June 2017.

[7] 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.

[8] G. J. Duck, X. Gao, and A. Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[9] K. Eldefrawy, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. 01 2012.

[10] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium*

on Microarchitecture, MICRO-47, page 190–202, USA, 2014. IEEE Computer Society.

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

[12] HexHive. Hexhive/retrowrite.

[13] jovanbulck. jovanbulck/nemesis.

[14] P. Koeberl, S. Schulz, V. Varadharajan, and A.-R. Sadeghi. Trustlite: A security architecture for tiny embedded devices. 04 2014.

[15] 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.

[16] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[17] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery.

[18] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. H. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, pages 156–168, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7:1–7:33, September 2017.

[20] 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.

[21] M. Salehi, D. Hughes, and B. Crispo. msbs: Static binary sanitization of bare-metal embedded devices for fault observability. pages 381–395. USENIX Association, 2020.

[22] sancus tee. sancus-tee/sancus-support.

[23] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. *Protected Software Module Architectures*, pages 241–251. 01 2013.

[24] 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.

[25] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Comput. Surv.*, 52(3), June 2019.

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

[27] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.