Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks

Hans Winderix
imec-DistriNet
KU Leuven
Leuven, Belgium
hans.winderix@kuleuven.be

Jan Tobias Mühlberg
imec-DistriNet
KU Leuven
Leuven, Belgium
jantobias.muehlberg@kuleuven.be

Frank Piessens
imec-DistriNet
KU Leuven
Leuven, Belgium
frank.piessens@kuleuven.be

***Abstract*—Recent controlled-channel attacks exploit timing differences in the rudimentary fetch-decode-execute logic of processors. These new attacks also pose a threat to software on embedded systems. Even when Trusted Execution Environments (TEEs) are used, interrupt latency attacks allow untrusted code to extract application secrets from a vulnerable enclave by scheduling interruption of the enclave. Constant-time programming is effective against these attacks but, as we explain in this paper, can come with some disadvantages regarding performance. To deal with this new threat, we propose a novel algorithm that hardens programs during compilation by aligning the execution time of corresponding instructions in secret-dependent branches. Our results show that, on a class of embedded systems with deterministic execution times, this approach eliminates interrupt latency side-channel leaks and mitigates limitations of constant-time programming. We have implemented our approach in the LLVM compiler infrastructure for the Sancus TEE, which extends the openMSP430 microcontroller, and we discuss applicability to other architectures. We make our implementation and benchmarks available for further research.**

***Index Terms*—Embedded systems; side-channel attacks; controlled-channel attacks; compiler hardening.**

1. Introduction

With the rise of the Internet of Things (IoT) and the increasing deployment of connected devices in critical domains such as healthcare and industrial sensing and actuation, embedded systems move into the focus of attacks. Many embedded systems are based on inexpensive low-power processors that lack sophisticated security mechanisms of high-end CPUs, but still need to execute cryptographic operations and computations on confidential data. To mitigate this attack vector, embedded Trusted Execution Environments (TEEs) [5], [12], [19], [21], [29], [32], [37] have been developed to provide enclaved execution for trusted code, which then runs in isolation from an untrusted context.

State-of-the-art TEEs typically adhere to an execution model that enables the interruption of trusted enclaves by untrusted code. Furthermore, low-end embedded TEEs are built upon CPUs with predictable instruction execution times, due to the absence of performance-enhancing techniques that introduce timing variations such as caches, branch predictors and out-of-order pipelines. As shown by Van Bulck et al. [43], this combination of features gives rise to a noiseless class of microarchitectural timing side-channels that rely on interrupt latency.

Listing 1: A secret-dependent branch that is vulnerable to a timing attack. The `FALSE` path consumes four cycles, the `TRUE` path only one.

```
1   CMP R12, R13    ; 1 cycle
2   JEQ .TRUE       ; 2 cycles
3 .FALSE
4   ADD R12, R13    ; 1 cycle
5   ADD R13, R14    ; 1 cycle
6   JMP  #.END      ; 2 cycles
7 .TRUE
8   SUB R14, R12    ; 1 cycle
9 .END
10
11 ; R12, R13 and R14 are live here
```

Listing 2: A secret-dependent branch with balanced execution times. Balancing execution times closes the start-to-end timing leaks but leaves the interrupt latency channel.

```
1   CMP R12, R13    ; 1 cycle
2   JEQ .TRUE       ; 2 cycles
3 .FALSE
4   ADD R12, R13    ; 1 cycle
5   ADD R13, R14    ; 1 cycle
6   JMP  #.END      ; 2 cycles
7 .TRUE
8   SUB R14, R12    ; 1 cycle
9   NOP             ; 1 cycle
10  NOP             ; 1 cycle
11  NOP             ; 1 cycle
12 .END
```

Start-to-End Timing Attacks. An attacker can extract secrets by observing the time it takes a system to perform some computation. Since the practical relevance of this attack was demonstrated [13], [28], timing attacks and mitigations have attracted widespread interest, in particular for cryptographic software. Listing 1 shows an assembly program for the popular TI MSP430 microcontroller. Assuming that the comparison in line 1 depends on a secret, this program is vulnerable to start-to-end timing attacks since the `.TRUE` and `.FALSE` branches can be distinguished based on their execution times, which leaks information about the secret.

To mitigate software-level attacks, including side-channels, techniques have been proposed to analyse and transform source code such that the resulting program is semantically equivalent to the input program but does not exhibit specific vulnerabilities. We refer to these techniques as *program hardening*. A technique that is often used to harden programs for low-cost embedded devices against start-to-end timing vulnerability is to manually equalise the execution time of secret-dependent branches, as illustrated in Listing 2. The `.TRUE` branch is padded with compensation code in the form of no-op instructions to match the execution time of the `.FALSE` branch. Although research [22] has shown that on high-end systems such balanced branch implementations can be broken by exploiting microarchitectural details of the underlying execution platform, it was believed that this is not the case for the simple microarchitectural designs of resource-constrained embedded devices.

Interrupt Latency Attacks. Van Bulck et al. [43] have demonstrated that simple microcontrollers are vulnerable to microarchitectural attacks too. They present Nemesis, the first controlled-channel attack [33], [47] against low-end embedded processors with a TEE, which exploits timing differences in the fetch-decode-execute operation of processors. Being able to precisely schedule interrupts (e.g., by programming a timer), and then abusing the microarchitectural property that interrupts are only served upon instruction retirement, an attacker can observe the interrupt latency.

As instructions can take a variable amount of CPU cycles to execute, Van Bulck et al. [43] illustrate convincingly that this observable variance is sufficient to distinguish the execution of secret-dependent branches and thereby extract application secrets from a vulnerable enclave. Balancing the start-to-end execution time, as done in Listing 2, does not provide protection. A Nemesis attacker is able to distinguish different executions based on the interrupt latency of individual instructions. The corresponding 2-cycle `jmp` (line 6) and 1-cycle `nop` (line 10) instructions demonstrate this vulnerability.

On high-end systems, the predominant strategy to harden software against information leakage through timing side-channels is by adhering to the constant-time policy [4], [8], [11], [16], [42]. Constant-time programming is a security policy that, among others, forbids secret-dependent jumps altogether, which effectively prevents attackers from inferring secrets from secret-dependent control-flow. Not having secret-dependent branches is effective as a countermeasure against interrupt latency attacks too.

However, compared to the balancing approach from Listing 2, constant-time transformations often result in larger and less efficient code. We illustrate this in Listing 3, which contains the if-converted code from Listing 1. If-conversion is a technique to convert control dependencies into data dependencies, which has been used before to eliminate secret-dependent branches [18], [35]. The 16-bit MSP430 architecture lacks predicated instructions, so we cannot use the technique proposed by Coppens et al. [18]. Therefore, in the example, we apply if-conversion based on Molnar et al. [35], which consists of the derivation of a mask from the branch condition, and the application of that mask in such a way that, although the instructions from both branches are executed, only the effects of one branch are taken into account. In our example, we compute the mask, either `0xFFFF` or `0x0000`, in register `R10`, and its inversion in `R11`. Before each computation, we store the original value of the result in the temporary register `R9`. Although the resulting code is free of timing leaks, it comes with a much higher cost with respect to code size and performance, compared to keeping but balancing the branches. This is due to the use of additional registers and extra instructions.

Listing 3: The if-converted code from Listing 1 results in secure but less efficient code.

```
1  ; Step 1) Compute masks
2
3  CMP R12, R13 ; if R12 is R13
4  MOV R2, R10  ;   bit1 of R2 is 1
5  RRA R10      ; shift right
6  AND #1, R10  ; R10 is 0 or 1
7  ADD #-1, R10 ; true mask
8  MOV R10, R11
9  XOR #-1, R11 ; false mask
10
11 ; Step 2) Apply if-conversion
12
13 ; .FALSE
14 MOV R13, R9
15 AND R10, R9  ; apply t mask
16 ADD R12, R13 ; ADD
17 AND R11, R13 ; apply f mask
18 BIS R9, R13  ; bitwise OR
19 MOV R14, R9
20 AND R10, R9  ; apply t mask
21 ADD R13, R14 ; ADD
22 AND R11, R14 ; apply f mask
23 BIS R9, R14  ; bitwise OR
24
25 ; .TRUE
26 MOV R12, R9
27 AND R11, R9  ; apply f mask
28 SUB R14, R12 ; SUB
29 AND R10, R12 ; apply t mask
30 BIS R9, R12  ; bitwise OR
```

Objective. Our objective is to automatically harden software for resource-constrained devices against leaking sensitive information via the interrupt latency channel with less overhead than if-converting the sensitive parts of the application. The approach must be automated to allow developers to create secure software using familiar programming abstractions. Timing leaks should be transformed out as late as possible in the compilation pipeline, preferably in the compiler backend. This renders the transformation source language independent, reduces the chance of later passes from breaking the protections, and it allows for earlier passes to be implemented without consideration of security concerns.

Contributions. Our contributions with this paper are as follows:

1) We present the first algorithm designed to protect embedded programs from sensitive information leakage through the recently disclosed interrupt latency controlled-channel attack [43]. Our algorithm is capable of hardening programs with non-trivial control-flow graphs by

supporting nested branches, statically bounded (nested) loops and function calls. Our algorithm is also effective as a countermeasure against start-to-end timing attacks.
2) We present an implementation of our algorithm for the MSP430 platform as a compiler pass in the LLVM [31] compiler infrastructure, supporting all MSP430 opcodes. On this deterministic microcontroller, our approach eliminates information leakage through interrupt latency side-channels completely. We target Sancus [37], an open-source TEE which extends the openMSP430 microcontroller. Our implementation is independent of high-level languages, eliminates vulnerable code introduced in early compiler passes, and can also be made independent of the target platform. We also discuss applicability to other architectures.
3) We conduct an empirical evaluation of our approach on a set of synthetic programs that exhibit a wide range of control-flow patterns, and on a set of third-party programs taken from five different sources.
4) Our results indicate that constant-time programming is not always the only defence to protect software from timing side-channel attacks, an assumption that seems to be widely accepted by the security community. We show that under certain systems models, alternatives to constant-time programming exist for hardening software against these attacks in a more efficient way.
5) We make our implementation and evaluation available at https://github.com/hanswinderix/sllvm.

Outline. We have structured this paper to first provide an intuitive description of our transformation in Section 2. In Section 3 we present the static analysis and program hardening phases of our transformation algorithm in detail; a complete example of how our approach affects a program is given in Section 4. We develop an LLVM-based implementation and elaborate on an experimental evaluation of our defence in Section 5. Finally we discuss related work and the difficulties of a comparative experimental evaluation of compiler-based defences against side-channel attacks in Section 6, and draw conclusions in Section 7.

2. The Defence

In this section we elaborate the objectives and assumptions for our hardening transformation, and provide an intuitive outline of how the approach works. Our objective is to automatically harden software for low-end processors against leaking secret-dependent control-flow through the interrupt latency channel. We aim to accomplish this by making latency traces of the victim's execution independent of secrets. Here, a latency trace is represented by the the list of latencies of the executed instructions along a path in the victim. We propose an algorithm that transforms a low-level program representation, such that instructions of alternative execution paths, that originate from a common secret-dependent branch, and that have the same distance from the branching instruction, have equal execution times. We realise this by intelligently inserting "dummy" instructions, which behave like no-ops (cf. Sect. 4 for a realistic example). Then, neither observing the execution time of different branches, nor repeatedly interrupting the program and measuring interrupt latency, allows an attacker to infer sensitive information.

2.1. Assumptions

Our algorithm requires semantic information regarding the sensitivity of program variables which must be provided by the application developer in the form of source code annotations. We assume that these annotations correctly express the sensitivity of the program variables. This information is automatically propagated throughout the compiler pipeline where we rely on established static analyses to infer more refined information such as behaviour about loops and the level of sensitivity of intermediate variables.

System Model. We restrict the scope of our defence to light-weight processors that are common in embedded devices and in the IoT. For our defence, these execution platforms must feature an instruction set with deterministic execution times. Our defence further requires the instruction set to allow for the construction of a dummy instruction for every possible latency class. We define such a dummy instruction to be an instruction without observable side effects besides its time to execute. With this model we specifically target low-end processors that lack advanced microarchitectural features such as caches, branch speculation, or out-of-order execution.

Our target processors can feature strong isolation concepts such as TEEs, e.g. Sancus [37] or TrustZone [5], that prevent untrusted code execution to directly access a protected application's memory address space. Here, controlled-channel attacks [47] may be mounted against the protected victim application, where the attacker leverages untrusted code execution to control system events such as interrupts. Issuing these attacker-controlled events then leads to the victim application leaking its secret state into the untrusted context.

Attacker & Leakage Model. In this paper we follow the attacker model of Van Bulck et al. [43], where an attacker is capable of scheduling interrupts to interrupt code that executes in a TEE so as to gather information about the internal state of this otherwise protected execution. We then assume a leakage model where program execution leaks its latency trace, i.e. the number of executed instructions and the latency of each individual instruction. Van Bulck et al. [43] have shown that this leakage model is applicable to stored program computers with a multi-cycle instruction set where either every individual instruction is uninterruptible and thus executes to completion or where instructions are abandoned to service pending interrupts and restarted when the corresponding Interrupt Service Routine (ISR) is done handling the interrupt.

2.2. Applicability

To demonstrate the generality of our work, we show that a number of popular processors, which have been embedded in tens of billions of devices, adhere to the model that is described in the previous section. More specifically, we discuss three low-end architectures that feature a two-stage pipeline, i.e, the next instruction is fetched while the

current is being executed. These are simple architectures without advanced features, such as branch speculation and caches, which renders the execution times of individual instructions predictable. We provide an implementation of our defence, which automates the program transformation, as a pass in the LLVM compiler backend for one of these architectures, the TI MSP430.

8-bit AVR (Atmel). AVR is a family of 8-bit microcontrollers developed by Atmel that targets low-end embedded systems. AVR is based on a modified Harvard architecture where special LPM instructions exist to access read-only data stored in program memory. AVR instructions [7] take between one and three clock cycles to execute. The number of CPU cycles that are consumed by a conditional branch instruction depends on whether the branch is taken or not. Special arrangements, such as disabling interrupts for a few cycles or address masking, have to be made for conditional branches to work around their non-determinism. The number of cycles required for the other instructions is completely determined at compile-time. Table 1 lists a possible selection of dummy instructions that covers the different latency classes of the AVR core instruction set. For a three-cycle dummy instruction, the compiler must reserve one ordinary and one pointer register which increases the register pressure as these registers cannot be used anymore for values with overlapping live ranges.

TABLE 1: A set of dummy instructions that covers the latency classes of the complete AVR core instruction set.

Instruction	Cycles	Size (bytes)
`SUBI R0, $0`	1	2
`CPSE R0, R0`	2	4
`LPM R0, Z`	3	2

16-bit MSP430 (Texas Instruments). The MSP430 is a microcontroller family from Texas Instruments built around a 16-bit processor and designed for low-cost and low-energy embedded applications. According to the MSP430xxx family user's guide [26] the execution of an MSP430 instruction is completely deterministic and takes between one and six clock cycles. The actual number of cycles depends on the addressing modes of the source and destination operands, not the instruction type itself. Jump instructions always take two cycles to execute, regardless of whether the jump is taken or not.

TABLE 2: A set of dummy instructions that covers the latency classes of the complete MSP430 instruction set.

Instruction	Cycles	Size (bytes)
`MOV #0, R3`	1	1
`MOV #42, R3`	2	1
`MOV 2(PC), R3`	3	2
`BIC #0, 0(R4)`	4	2
`MOV @R4, 0(R4)`	5	2
`MOV 0(R4), 0(R4)`	6	3

Table 2 lists a possible selection of dummy instructions that covers the different latency classes of the MSP430 instruction set. We present and evaluate an implementation of our defence for this processor in Sect. 5.

32-bit Cortex-M23 (ARM). The ARM Cortex-M family is a group of 32-bit processor cores with a microcontroller profile, optimised for low-cost and low-energy microcontrollers [6]. The Cortex-M23 core supports the ARMv8-M baseline instruction set and can be combined with TrustZone [5] technology. Instructions take between one and five clock cycles to execute, except for the integer multiplication and division instructions, whose execution time is implementation-dependent. As with the AVR case, a conditional branch consumes one or two CPU cycles depending on whether the branch is taken or not. Thus, similar measures have to be taken to neutralise the non-determinism of this instruction. The number of cycles required for the other instructions is completely determined at compile-time. The Cortex-M23 core does not implement the IT instruction and thus does not support conditional execution. Table 3 lists a possible selection of dummy instructions that covers the different latency classes of the ARMv8-M baseline instruction set. The cycle counts are based on a system with zero wait-states on the AHB bus, no single-cycle I/O port, a hardware multiplier speed of 32 cycles and a hardware divider speed of 34 cycles. For the 4-cycle, 32-cycle and 34-cycle dummy instructions, the compiler must reserve one register, for the 5-cycle dummy instruction three registers. This increases register pressure as these registers cannot be used anymore for values with overlapping live ranges.

TABLE 3: A set of dummy instructions that covers the latency classes of the Armv8-M baseline instruction set.

Instruction	Cycles	Size (bytes)
`MOV R0, R0`	1	2
`B <label>`	2	2
`B.W <label>`	3	2
`BXNS R0`	4	2
`LDM R0, R1, R2`	5	2
`MULS R8, R8, R0`	32	2
`UDIV R8, R8, R0`	34	4

2.3. Alignment Algorithms

Alignment algorithms are a class of algorithms that perform a program transformation such that corresponding instructions of alternative execution paths have equal execution times. Instructions correspond when they originate from a common branch and when they are located at the same distance from that branch. Alignment algorithms perform the following operations:

Equalise Path Lengths. Convert the program's control-flow graph (CFG) into a functionally equivalent one such that, for every secret-dependent branch, every path through the corresponding CFG region of that branch has the same number of basic blocks.

A basic block is a sequence of instructions that is always entered at the beginning and exited at the end of that sequence. Equalising path lengths can be accomplished, e.g., by inserting dummy basic blocks or by duplicating basic blocks.

Algorithm 1 Algorithm for computing the level structure based on a breadth-first traversal starting from the region's entry basic block. Each path through the region is assumed to contain an equal number of basic blocks.

```
1:  function ComputeLevelStructure(Entry : BasicBlock, Exit : BasicBlock)
2:      CurLevel ← 0
3:      Result[CurLevel] ← {Entry}
4:      while Exit ∉ Result[CurLevel] do
5:          CurLevel ← CurLevel + 1
6:          Result[CurLevel] ← ∅
7:          for all u ∈ Result[CurLevel − 1] do
8:              for all u → v do
9:                  Assert(¬ Visited(v))
10:                 Result[CurLevel] ← Result[CurLevel] ∪ {v}
11:             for all u → v do
12:                 Visited(v) ← true
13:     Assert(|Result[CurLevel]| = 1)
14:     return Result
```

Compute the Level Structure. For every secret-dependent branch, compute the level structure of the corresponding CFG region. The level structure of a CFG region is a partition of the region's basic blocks into subsets (levels) that have the same distance from the region's entry block. Algorithm 1 shows how to compute the level structure based on a breadth-first traversal of an acyclic CFG with equal path lengths. The equal path length property guarantees that, during the traversal, no basic block is visited at more than one level, and that the region's exit block is reached for all paths simultaneously.

Align Basic Blocks. For every secret-dependent branch, equalise the execution times of corresponding instructions, such that basic blocks at the same level of the level structure have the same latency trace. This can be accomplished, e.g., by inserting dummy instructions, reordering instructions or strength reducing (or enhancing) instructions, where the latter transformation rewrites instructions into functionally equivalent ones, but with different execution times.

2.4. A Naive Algorithm

We now consider a naive alignment algorithm. Figure 1a depicts an acyclic CFG where basic block A terminates in a secret-dependent branch instruction. Step 1, equalising the path lengths, can be accomplished by serialising every possible path through the CFG based on a topological order of the basic blocks, such that each level either contains an empty dummy basic block or a clone, i.e., a copy of another basic block. This is illustrated in Figure 1b. Step 2, the computation of the level structure, can then be done with Algorithm 1. For step 3, the level-wise alignment of the basic blocks, it suffices to add dummy instructions to the empty dummy basic blocks such that their latency trace matches the latency trace of the clones.

This approach is suboptimal for several reasons. First, opportunities for sharing basic blocks between different paths are disregarded. Second, compensation code for all the paths in an inner region is added to the paths that are not part of that region. As already observed in prior work by Agat [3], this unnecessarily increases the size of the resulting code. Third, corresponding instructions with equal execution times are not taken advantage of.

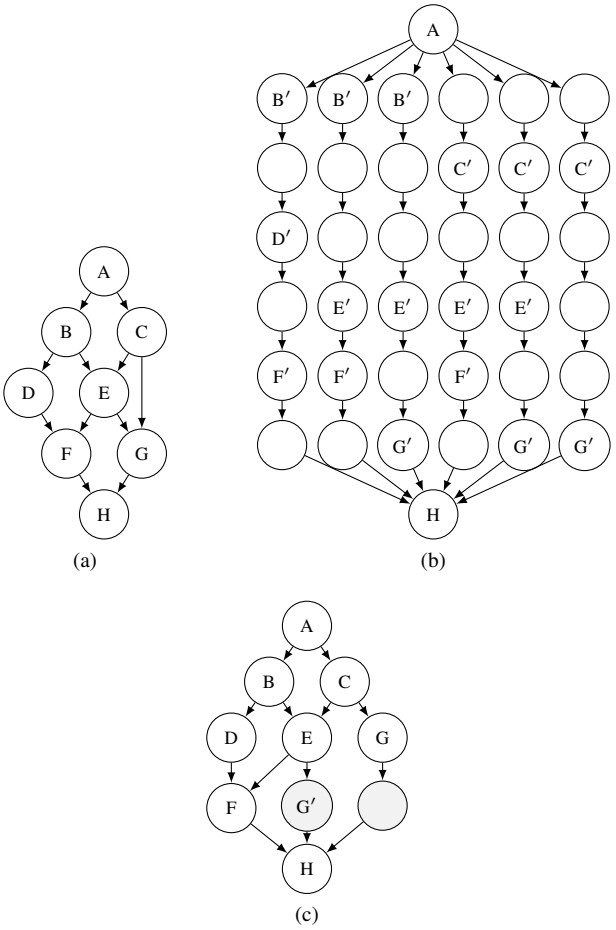

Figure 1: Two alternative strategies to equalise path lengths. For all these CFGs, basic block A ends in a secret-dependent branch. A vulnerable CFG (a), length equalisation based on topological order (b), and a more efficient approach (c).

To improve on these issues, operations that negatively impact performance, such as duplicating basic blocks and inserting dummy instructions, should be kept to a minimum. Finding this minimum is a challenging optimisation problem.

2.5. A More Efficient Algorithm

We now present a high-level overview of our approach which performs the three operations from Section 2.3 in a single pass. We also take advantage of opportunities to reduce the duplication of basic blocks and the insertion of dummy instructions. Furthermore, our algorithm supports function calls, nested branches and loops with statically known bounds. The algorithm can be extended to handle loops with statically unknown bounds, as long as the termination condition does not depend on sensitive information. Supporting this requires analysing the loop condition and the operations on the induction variable(s) to be able to duplicate them in the compensation code.

Computing the Level Structure. Instead of sequentially *(1)* equalising path lengths, *(2)* then computing the level structure, and *(3)* then performing a level-wise alignment of the basic blocks, our algorithm aligns the basic blocks of a level as soon as the members of that level have been determined. A single operation computes the next level of the level structure and equalises path lengths based on a breadth-first traversal of the CFG.

Empty Basic Blocks. When computing the next level of the level structure, one of the successors of a basic block of the current level may be identified as the region's exit block, while not being the case for (some of) the other blocks of the current level. This means that the corresponding path is shorter than (some of) the other paths. This situation is illustrated in Figure 2a. To remedy this, we lengthen the shorter path by inserting an empty basic block right before the exit block, illustrated in Figure 2b. When more than one path prematurely exits the sensitive region, it suffices to insert a single empty block that can be shared amongst all lengthened paths.

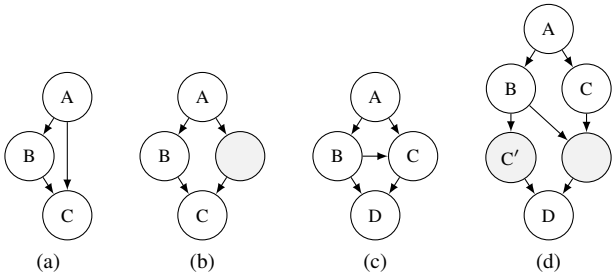

Figure 2: For all these CFGs, basic block A ends in a secret-dependent branch. The vulnerable CFG in (a) is hardened in (b) by adding an empty block, to be aligned later with basic block B. Another vulnerable CFG in (c) is hardened in (d) by cloning basic block C in C′ and by adding a shared empty block as the successor of both B and C, to be aligned later with C′.

Cloning Basic Blocks. When performing a breadth-first traversal of a CFG region with unequal path lengths, it is possible that a basic block is visited more than once. Such a block can not be aligned multiple times, as this will break a previously hardened level. Our algorithm deals with this situation by cloning already visited blocks, as illustrated in Figure 2d.

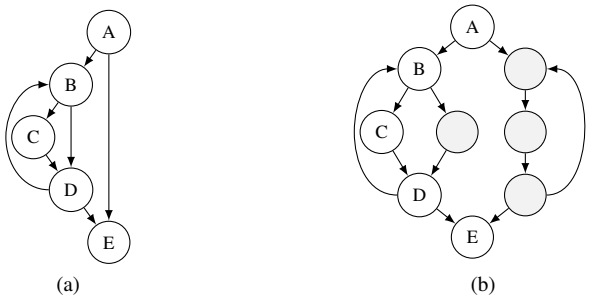

Figure 3: For all these CFGs, basic block A ends in a secret-dependent branch. The vulnerable CFG in (a) is hardened in (b) by adding a dummy loop in the right path to compensate for the loop in the left path.

Loops. When our algorithm detects a natural loop, it first recursively hardens the CFG region that corresponds to that loop. Then, the latency trace of the hardened loop is computed which is used to create compensating dummy loops (cf. Figure 3). Programs with sensitive loops that are not statically bounded are rejected.

Aligning a Set of Basic Blocks. In his seminal work on transforming out timing leaks, Agat [3] proposes a technique to align a set of basic blocks by performing a crosswise padding of every basic block with the low-slice of the others. A downside of cross-copying is that equal execution times of corresponding instructions are not taken advantage of, which would avoid inserting unnecessary dummy instructions.

Figure 4 illustrates our alignment strategy. For every basic block, we maintain an instruction iterator, which is initialised to point to the first instruction of the basic block it represents. Each of these instruction iterators in turn takes on the role of the *reference iterator*. The execution times of corresponding instructions are compared to the execution time of the reference instruction. When the execution times match, the comparing iterator is advanced, otherwise a compensating dummy instruction is inserted. After checking all corresponding instructions, the reference iterator is advanced.

Trusted Function Calls. An iterator pointing to a call instruction temporarily takes on the role of the reference iterator. Then, a hardened and a dummy version of the callee are generated, the call is replaced by a call to its hardened version, and a call to the dummy version is inserted in the other paths. Finally, the reference iterator is advanced and its temporary role ends.

Putting It All Together. Applying our path length equalisation strategy to the vulnerable CFG from Figure 1a demonstrates the impact of the chosen strategy. As illustrated by Figure 1c, in order to equalise the path lengths of the vulnerable CFG, our algorithm needs to add only two blocks: one dummy block and one clone. The topological order-based approach from Figure 1b requires 11 clones and 19 dummy blocks.

Similarly, our strategy to align a set of basic blocks limits the number of instructions of the basic blocks from Figure 4 to four, and only adds four dummy instructions in total. Applying cross-copying would result in basic blocks with nine instructions, or 19 additional instructions.

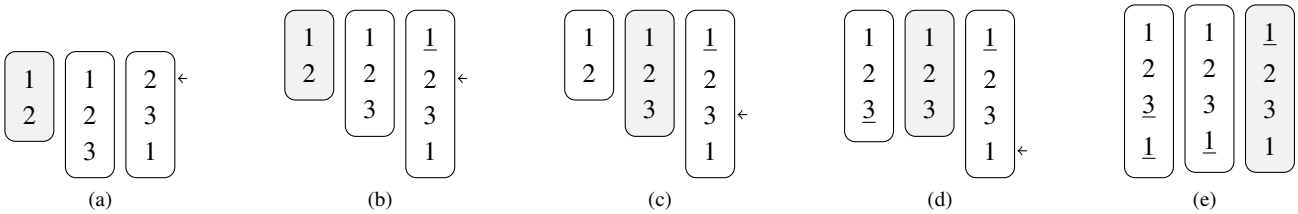

Figure 4: Aligning a set of basic blocks from a vulnerable (a) to a hardened set (e). Shaded blocks represent reference blocks, arrows indicate where the instruction iterators are pointing to, dummy instructions are underlined.

3. The Algorithm

In this section, we present our algorithm in more detail. We decompose it in a static analysis and a program hardening phase; we focus on the latter.

3.1. Static Analysis

The static analysis phase consists of a number of compile-time analyses that conservatively approximate the run-time behaviour of the program under analysis. Static analysis informs the program hardening phase. Note that this paper does not aim to advance the state-of-the-art in static analysis, and we consider this orthogonal research. At the expense of less efficient output programs, an algorithm can reduce its reliance on static analysis; and at the expense of more work for the programmer, an algorithm can rely more on programmer annotations. These approaches are taken in related work, such as in [18].

First of all, the *control-flow analysis* computes the program's control-flow graph (CFG). A node in the CFG represents a basic block, an edge stands for a possible flow of execution from one block to another. The *dominance analysis* computes the dominance and post-dominance relations between the CFG nodes. The outcome of this analysis, the dominator tree, is useful, e.g., to determine if a basic block is control-dependent of a secret-dependent branch, or to find the exit block of a sensitive CFG region. *Loop analysis* uses the CFG to identify natural loops and special loop blocks, such as the loop pre-header, header, latch and exit blocks. Loop analysis also computes the trip count of sensitive loops. *Reaching definitions analysis* (RDA) collects information about which assignments reach which instruction operands. Finally, the *sensitivity analysis* is responsible for the identification of the different sensitive regions in the CFG, based on the security annotations from the developer and the RDA results.

3.2. Program Hardening

Algorithm 2 contains simplified pseudo-code for our program hardening phase, which consists of a number of elementary operations that we discuss in the following.

AlignSensitiveBranch. The parameters of this operation are the entry and the exit blocks of the sensitive branch region to align. This operation first invokes *ComputeNextLevel* to retrieve the first level of the level structure based on its entry block, and then traverses the region in level-order until it reaches the exit block. Every iteration deals with two possible cases. First, if a loop has been detected, all contained regions in the loop are aligned, and a dummy loop is inserted before every basic block from the current level. Otherwise, the basic blocks in the current level are aligned with each other, as described in section 2.

ComputeNextLevel. The operation *ComputeNextLevel* is at the heart of our alignment algorithm. It determines the next level to align, equalises paths lengths, clones basic blocks and detects loops. The end of a sensitive branch region is reached when the given level is the last level of the region's level structure. This is the case when every basic block of that level has the exit block as its single successor (line 13). A loop is found when one of the successor blocks is the header of a natural loop (lines 14-18). To determine the next level, *ComputeNextLevel* computes the union of the direct successors of all basic blocks belonging to the given level (lines 19–29). Three cases are dealt with. First, an empty basic block is added to every path that prematurely reaches the exit block, making the empty block an element of the next level. Second, already aligned successor blocks are cloned, and each clone is added to the next level. Third, if the first two cases do not apply, it is the successor block itself that is added to the next level.

AlignContainedRegions. This operation aligns all the contained regions of the given loop. First it recursively invokes *AlignContainedRegions* for every directly nested loop. Then, it recursively invokes *AlignSensitiveBranch* for every directly nested branch. The recursive nature of this operation makes sure that eventually all the contained regions, at any nesting level, are aligned.

InsertDummyLoops. This operation inserts a dummy loop before every basic block that belongs to the given level, such that every dummy loop produces the same latency trace as the given loop. Note that the caller of this operation must make sure that the given loop is already aligned. We define the footprint of an aligned loop as the latency trace of a single iteration. By definition, every iteration of an aligned loop produces the same latency trace. We represent a footprint as a list of basic blocks.

InsertDummyLoops iterates over all the basic blocks of the given level, (line 38), inserts a dummy block for each footprint block (lines 39-44, 59), and aligns each dummy block with the corresponding footprint block (line 51). For every loop header block that is encountered along the way, a compensating loop preheader is inserted, and instructions are generated that initialise the induction

Algorithm 2 A simplified version of the alignment algorithm

```
1: procedure AlignSensitiveBranch(Entry : BasicBlock, Exit : BasicBlock)
2:     (Loop, Level) ← ComputeNextLevel({Entry}, Exit)
3:     while Level ≠ ∅ do
4:         if Loop = None then
5:             AlignBasicBlocks(Level)
6:         else AlignContainedRegions(Loop)
7:             Level ← InsertDummyLoops(Level, Loop)
8:         (Loop, Level) ← ComputeNextLevel(Level, Exit)
9: function ComputeNextLevel(Level : List of BasicBlock, Exit : BasicBlock)
10:    Loop ← None
11:    NextLevel ← ∅
12:    Successors ← Flatten({ Successors(x) | x ∈ Level})
13:    if ∃x ∈ Level : |Successors(x)| > 1 ∨ First(Successors(x)) ≠ Exit then
14:        LoopHeaders ← {x | x ∈ Successors, IsLoopHeader(x)}
15:        if LoopHeaders ≠ ∅ then
16:            Header ←First(LoopHeaders)
17:            Loop ←LoopFor(Header)
18:            NextLevel ← Successors \ {Header})
19:        else Empty ← CreateBasicBlock
20:            for all Block ∈ TopologicalSort(Level) do
21:                for all Successor ∈ Successors(Block) do
22:                    if Successor = Exit then
23:                        Connect(Block ↛ Exit, Block → Empty → Exit)
24:                        NextLevel ← NextLevel ∪ {Empty}
25:                    else if IsAligned(Successor) then
26:                        Clone ← CloneBasicBlock(Original(Successor))
27:                        Connect(Block ↛ Successor, Block → Clone)
28:                        NextLevel ← NextLevel ∪ {Clone}
29:                    else NextLevel ← NextLevel ∪ {Successor}
30:                BreakCycles(Block, Level)
31:    return (Loop, NextLevel)
32: procedure AlignContainedRegions(Loop)
33:    ForEach(Loops(Loop), λx.AlignContainedRegions(x))
34:    ForEach(SensitiveBranches(Loop), λx.AlignSensitiveBranch(x))
35: function InsertDummyLoops(Level : List of BasicBlock, Loop)
36:    NextLevel ← {LoopExit(Loop)}
37:    for all Block ∈ Level do
38:        LoopHeaders ← ∅, Prev ← None
39:        for all Ref ∈ Footprint(Loop) do
40:            BB ← CreateBasicBlock
41:            if Prev ≠ None then
42:                Connect(Prev ↛ Block, Prev → BB)
43:            else
44:                ForEach(Predecs(Block), λx.Connect(x ↛ Block, x → BB))
45:            if IsLoopHeader(Ref) then
46:                PreHeader ← BB
47:                BB ← CreateBasicBlock
48:                Connect(PreHeader → BB)
49:                Push(LoopHeaders, BB)
50:                PushAndInitInductionRegister(PreHeader)
51:            AlignBasicBlocks({Ref, BB})
52:            if IsLoopLatch(Ref) then
53:                Exit ← CreateBasicBlock, Header ← Pop(LoopHeaders)
54:                Connect(BB → Header, BB → Exit)
55:                UpdateAndCompareInductionRegister(BB)
56:                PopInductionRegister(Exit)
57:                BB ← Exit
58:            Connect(BB → Block)
59:            Prev ← BB
60:        NextLevel ← NextLevel ∪ {Exit}
61:    return NextLevel
```

register after pushing the current value of that register on the stack (lines 45-50). When encountering the loop latch block, a back edge to the dummy header is created, and a dummy loop exit block is inserted. Instructions to update and compare the induction register, and to restore the previous value of the induction register are generated (lines 52-57). The return value of this operation is the last aligned level so far, which consists of all the loop exit blocks (lines 36, 60).

3.3. Complexity

Let us briefly consider the worst case performance of our algorithm, which occurs under two conditions. First, all corresponding instructions before hardening must have different execution times, in which case the performance of our transformation degenerates to that of the cross-copying method from Agat [3]. Second, every node (in the first half) of the CFG must end in a two-way branch. These conditions will result in a blow-up that grows exponentially in the number of levels, which negatively impacts both code size and execution time. It is possible to mitigate the size blow-up by factoring out common dummy code into dedicated functions. We leave this for future work.

4. A Comprehensive Example

Listing 4: Comprehensive example program in C, featuring nested loops, nested `if` statements and function calls.

```
1  int bar(int a, int b);
2
3  int foo(secret int a, int b) {
4    int result = 3;
5
6    if (a < b) {
7      for (int i=0; i<3; i++) {
8        int j, r;
9
10       if (a == 12)
11         bar(i, j);
12
13       for (j=0; j<3; j++)
14         r = bar(i, j);
15
16       if (b == 12)
17         bar(r, r*r);
18     }
19   }
20   return result;
21 }
```

We present an example with non-trivial control-flow to demonstrate the working of our defence. Listing 4 contains an excerpt from the `ifthenloooplooptail` benchmark program (cf. Sect. 5). The function `foo` contains function calls, nested if-statements and nested loops. The `secret` annotation in the function signature of `foo` signals our implementation that parameter `a` of `foo` contains sensitive information.

Figures 5 and 6 respectively show the vulnerable CFG and the hardened CFG of `foo`. In hardened programs, symbol names with the `_nds_` and the `_ndd_` prefixes represent hardened and dummy versions of the corresponding unhardened function.

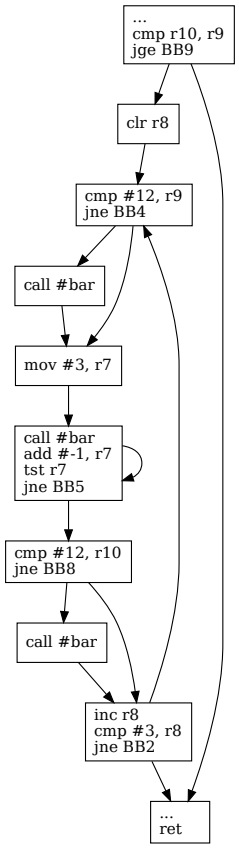

Figure 5: Vulnerable CFG of Listing 4.

5. Implementation and Evaluation

In this section we discuss our reference implementation. We also present an experimental evaluation, where we consider effectiveness and performance.

5.1. Implementation

We implemented the algorithm as a MachineFunction pass in the LLVM [31] MSP430 back-end, where we systematically traverse all assembly level functions to align the sensitive branches. By leveraging LLVM, we avoid having to implement a whole range of compiler infrastructure ourselves. Moreover, the target independent code generator framework makes it possible to write target independent algorithms.

We chose the openMSP430 as our target architecture because it presents a small and stable open-source execution platform, for which a TEE has been implemented through the Sancus extensions [37]. The Nemesis attack has been demonstrated on this architecture [43], and orthogonal research to detect vulnerable programs [40] and to defend against Nemesis with hardware modifications [14] have been presented based on the openMSP430 and Sancus. As we project in Sect. 2, implementing our defence for other architectures such as AVR or certain ARM cores should be possible.

To be secure, the program hardening phase must be informed by sound analyses. It would not be secure, for example, if the secret-dependent condition of a branch is analysed as secret-independent. Hence, we rely on

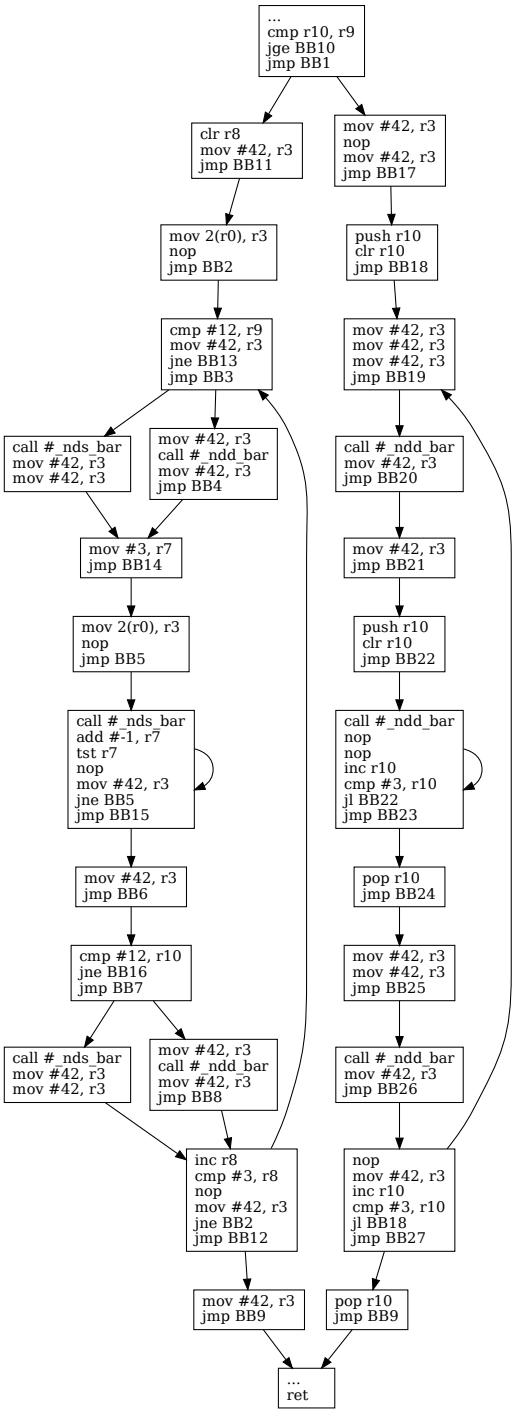

Figure 6: Hardened CFG of Listing 4.

sound analyses only, also in the presence of pointers and aliases. Therefore, some valid C programs will be rejected. For instance, our implementation is not able to compute the trip count of all statically bounded loops. However, our implementation is complete enough to accept all the programs of our two benchmark suites. Sound over-approximations of the analysis do have a performance impact. For example, our implementation might incorrectly classify insensitive branches as sensitive. This is secure but it obviously has a negative impact on performance.

Our implementation of the RDA and sensitivity analyses leverage the LLVM compiler infrastructure by building upon its control-flow, dominance and loop analyses. We kept these data-flow analyses simple by imposing strict well-formedness criteria (e.g. to be able to detect the loop trip count) and by providing sometimes imprecise (but sound) information (e.g. concerning the sensitivity of intermediate values). An improved static analysis is out of scope of this paper.

Adding annotations to the source code that reflect the intended security semantics is the task of the developer by indicating which function parameters are secret (cf. Sect. 4). To this end, we modified Clang, the LLVM front-end for the C family of programming languages. We conservatively consider global variables to be secret, do not distinguish between the secrecy of pointers and the memory they point to, neither do we allow to declassify secret variables. Improving the expressiveness of our annotation model will have a positive impact on performance. We leave this as future work.

5.2. Benchmark Suite

An existing suite of annotated benchmark programs with timing side-channel vulnerabilities would be ideal for evaluating our implementation. Unfortunately, to the best of our knowledge, there is no such suite. For this reason, we composed a meaningful benchmark suite ourselves, that consists of *(1)* a set of twelve synthetic programs, featuring a wide range of control-flow patterns, and specifically designed for this kind of evaluation, and *(2)* ten realistic third-party programs that have been used before in the evaluation of related work.

The twelve synthetic benchmark programs we created are derived from the unit tests that we used to test our implementation. These unit tests, and consequently the derived benchmark programs, exhibit diverse CFG patterns in order to achieve high code coverage, which is necessary to assess the effectiveness of our approach. These CFG patterns range from simple triangle-shaped branches, over branches with loops, to multi-level nested branches containing loops and function calls.

Furthermore, we selected ten third-party benchmark programs from five independent sources: six benchmark programs from related academic work [34], [43], two programs from a competitive embedded benchmark suite [25], one program from the Botan cryptographic library [1], and one program from the runtime of Texas Instruments' MSP430-GCC-OPENSOURCE [27] compiler. First, in [34], Mantel and Starostin experimentally evaluate four source-to-source transformations for mitigating timing side-channel vulnerabilities. For this evaluation, four Java programs programs from different domains with different degree of sophistication were selected; we ported these programs to C: *(1)* square-and-multiply modular exponentiation from RSA, *(2)* computation of a share's value, *(3)* Kruskal's algorithm for calculating the minimum spanning tree (MST) of a graph, and *(4)* modular multiplication from the IDEA cipher. Second, we included code from the two Sancus case studies from the Nemesis [43] paper: *(1)* a password comparison routine from Texas Instrument's MSP430 serial Bootstrap Loader (BSL) implementation and *(2)* a secure I/O application that prevents an untrusted operating system from reading

a secret PIN code. Third, we selected the two programs from the competitive benchmark suite from Texas Instruments [25] which do not consist of straight-line code and thus have branches: the 8-bit and the 16-bit switch case programs. Fourth, the Botan cryptographic library [1] provides an implementation of the key schedule function of the Twofish block cipher with a control dependency on the secret key, which we also used for our evaluation. Finally, since integer multiplication is not natively supported by all MSP430 microarchitectures, a software routine is available in the the compiler runtime. Our evaluation includes a slightly modified version of the integer multiplication routine provided by Texas Instruments' MSP430-GCC-OPENSOURCE toolchain [27].

The majority of our benchmark suite consists of non-cryptographic code. Real-world cryptographic code is typically written in a constant-time fashion, and such code does not contain secret-dependent control-flow. For instance, Wu et al. [46] analysed 19,708 lines of C/C++ code from real-world cryptographic libraries and identified only a single secret-dependent if-statement across these libraries. Our benchmark suite does contain primitive operations that are often used in cryptographic code: integer multiplication, modular multiplication, and modular exponentiation. Non constant-time implementations of these primitives can render cryptographic code vulnerable to side-channel attacks.

5.3. Experimental Setup

We conducted our experiments on a Dell Optiplex 9020 desktop with a Haswell quad-core Intel i7-4790 CPU, clocked at 3.6 GHz, with 16 GB of RAM, and running Ubuntu 20.04 with a generic 64-bit Linux 5.4.0 kernel. We compiled our experiments with version 0.9b of Security Enhanced LLVM (SLLVM) [45], which is adapted from LLVM 13.0.0. We assembled and linked the experiments with version 8.3.0.16 of the Texas Instruments MSP430-GCC-OPENSOURCE C compiler toolchain. We linked against the SLLVM development versions of the Sancus [37] support library and the Sancus compiler runtime. We ran the experiments on the cycle-accurate Sancus simulator which is based on the openMSP430.

Every benchmark program is similarly structured: the `main` function invokes the entry function of a Sancus enclave that performs a sensitive computation. To achieve maximal *branch and path coverage*, the entry function is invoked several times, each time with different but carefully chosen input parameters in order to trigger different execution paths. Each such invocation corresponds to what we will refer to as an experiment.

To normalise the results, we established a *baseline* where we performed each experiment with our defence disabled. To determine the effectiveness and the overhead of our approach, we then performed each experiment with our defence enabled. The results that we report are based only on the code that is executed in the enclave.

Finally, to obtain cycle-accurate timing measurements, we ran the compiled benchmarks on the Sancus simulator, which produces a waveform file that represents the state of the CPU during simulation. From the waveform file, we then extracted the latency trace for each experiment. Since the Sancus instruction execution times are deterministic, this is exactly as on real hardware.

5.4. Experimental Results

We verified the effectiveness of our defence by a manual inspection of the latency traces of the experiments. For all experiments, we are able to reveal the secret-dependent control-flow prior to hardening, demonstrating that every benchmark is indeed vulnerable to the Nemesis attack. Hardening the code with our defence effectively renders all experiments observationally equivalent regarding instruction latency. Thus, all interrupt latency and start-to-end timing vulnerabilities are eliminated. For instance, Figure 7 and Figure 8 depict the latency traces for the three experiments of the diamond benchmark before and after hardening, respectively.

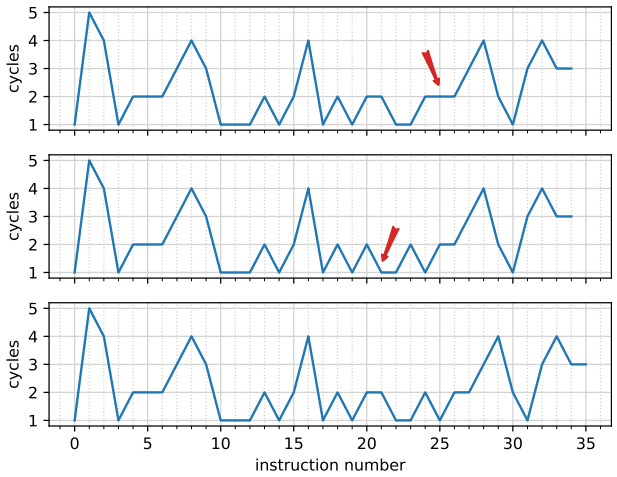

Figure 7: Latency traces of the vulnerable experiments for the diamond benchmark. Each figure corresponds to the latency traces of one experiment. The arrows point to instructions that are sufficient to distinguish the three experiments.

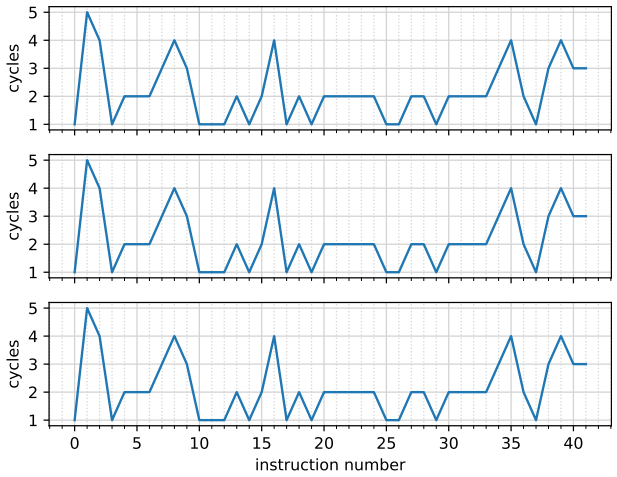

Figure 8: Latency traces of the hardened experiments for the diamond benchmark. Each figure corresponds to the latency trace of one experiment. The experiments cannot be distinguished anymore.

TABLE 4: Synthetic benchmark suite. Each row contains the performance results of one benchmark. Columns 2 and 3 contain the absolute measurements for the vulnerable baseline, columns 4, 5 and 6 contain the overhead of our defence relative to the vulnerable baseline. Columns 3 and 5 contain a value for each experiment, where an experiment corresponds to a unique flow of execution. Column 6 contains the overhead relative to the execution time of the longest execution path.

Benchmark	**Vulnerable Baseline**		**Overhead of Hardening**		
	Size (bytes)	Execution time (cycles)	Size	Execution time	Execution time (relative to optimum)
call	302	112, 91	1.09x	1.05x, 1.30x	1.05x
diamond	284	102, 101, 103	1.16x	1.13x, 1.14x, 1.12x	1.12x
fork	264	90, 91	1.06x	1.07x, 1.05x	1.05x
ifcompound	384	370, 371, 372	1.06x	1.02x, 1.02x, 1.02x	1.02x
ifthenloop	284	143, 96	1.27x	1.19x, 1.77x	1.19x
ifthenloopif	342	179, 108	1.38x	1.60x, 2.66x	1.60x
ifthenlooploop	308	378, 101	1.54x	1.36x, 5.08x	1.36x
ifthenlooplooptail	350	387, 387, 113	1.63x	1.25x, 1.25x, 4.27x	1.25x
indirect	274	95, 97	1.18x	1.19x, 1.16x	1.16x
loop	400	2841	1.06x	1.02x	1.02x
multifork	290	92, 100, 96, 99	1.19x	1.18x, 1.09x, 1.14x, 1.10x	1.09x
triangle	266	92, 94	1.09x	1.09x, 1.06x	1.06x
Geometric mean			**1.21x**	**1.32x**	**1.15x**

The performance results of the synthetic benchmarks are summarised in Table 4. Each row in this table contains the results of one benchmark program. The second and the third column contain the absolute measurements of the vulnerable baseline in terms of code size and execution time. Column four contains the overhead in code size of each compiled benchmark program relative to the code size of the compiled vulnerable baseline. Column five contains the overhead in execution time for each hardened experiment relative to its corresponding vulnerable baseline experiment. As said before, each benchmark is used for one or more experiments, which explains the multiple values in columns three and five,

Program hardening techniques that make sure that the execution time of secret-dependent code is constant, such as ours and the if-conversion technique [18], [35], hit a fundamental performance limit which is determined by the execution path with the longest execution time. Indeed, to ensure that the different execution paths through a secret-dependent region have equal execution times, it is impossible to do better than this optimum. This is the reason for the sometimes large differences in overhead between experiments of the same benchmark. For instance, the overhead of the second experiment of the *ifthenlooploop* benchmark (5.08x) corresponds to the empty *then* execution path through the secret-dependent region. After hardening however, this execution path is padded with a large number of dummy instructions to match the much longer alternative path. The loops further amplify this effect. Therefore, we believe that it is sometimes better to report the overhead in execution time relative to the execution time of the longest execution path. The last column of Table 4 contains this metric.

We summarised the performance results of the third-party benchmark programs in Table 5. To be able to compare our technique with constant-time programming, we created a straight-line version for each of these programs by manually performing if-conversion at source code level according to the technique proposed by Molnar et al [35]. To prevent the compiler from introducing branches again, and thus undoing the carefully if-converted code, we compiled all versions of this second batch of benchmarks with optimisations disabled. On average, the overhead of the third-party benchmark suite is on par with the overhead of the synthetic suite. Overall, if-conversion performs worse than our technique, although for some benchmarks (keypad, mulhi3, mulmod8, sharevalue, and twofish), our approach produces slightly larger binaries, which is a consequence of compiling with optimisations disabled.

Based on the empirical results of these experiments, we conclude that on average our approach results in an increased code size of 22% and in an increased execution time of 17%. On the target architecture, our transformation also outperforms if-conversion, for which we observed overheads of 35% and 76% for the third-party benchmark programs. We believe our overheads are reasonable given the offered security. As part of future work, we would like to investigate some optimisations that can further reduce this overhead.

6. Related Work

As Kocher points out in his seminal work on timing side-channels [28], making software run with fixed timings is hard. Unexpected timing variations can be introduced by performance improving techniques such as compiler optimisations and microarchitectural features. A long line of research [2], [24], [33], [38], [39], [47], [48] has successfully demonstrated attacks that exploit features found in modern out-of-order processors, including virtual memory, caches and branch predictors. Brumley and Boneh [13] show the feasibility of remote timing attacks.

6.1. Compile-Time Defences

Transforming out timing leaks by balancing secret-dependent branches was first proposed by Agat in [3], where a sound transformational security type system for a simple imperative language is described. The transformation equalises the execution time of sensitive branches by cross-copying skip instructions. Kopf and Mantel [30]

TABLE 5: Third-party benchmark suite. Each row contains the performance results of one benchmark. Columns 2 and 3 contain the absolute measurements for the vulnerable baseline. Only the path with the longest execution time is reported. Columns 4 and 5 contain the overhead of our defence relative to the vulnerable baseline, columns 6 and 7 the overhead of if-conversion.

Benchmark	**Vulnerable Baseline**		**Overhead of balancing**		**Overhead of if-conversion**	
	Size (bytes)	Execution time (cycles)	Size	Execution time	Size	Execution time
bsl	394	984	1.12x	1.20x	1.27x	1.47x
keypad	672	1119	1.28x	1.56x	1.24x	1.81x
kruskal	634	2460	1.14x	1.08x	1.16x	1.24x
modexp2	702	23537	1.05x	1.31x	1.05x	1.32x
mulhi3	416	904	1.37x	1.59x	1.34x	2.01x
mulmod8	482	425	1.49x	1.07x	1.40x	1.36x
sharevalue	480	3398	1.06x	1.04x	1.05x	1.07x
switch16	402	115	1.41x	1.09x	2.29x	4.65x
switch8	402	115	1.41x	1.09x	2.29x	4.65x
twofish	8872	92745	1.06x	1.00x	1.02x	1.02x
Geometric mean			**1.23x**	**1.19x**	**1.35x**	**1.76x**

propose a unification-based approach to balance instructions. Dewald et al. [20] apply Agat's idea in practice, with a non-transformational security type system to detect unbalanced branches on the AVR platform.

In comparison, our algorithm leads to more efficient code than the technique described by Agat, as hardened programs typically consist of fewer dummy instructions. We also believe it is important to separate the concerns of transformation and verification. Not requiring a compiler pass to be formally verified, makes it possible to realise complex transformations. A simple and elegant, but formally proven verifier can then provide strong guarantees that the resulting code is free of timing leaks.

The current trend to counter timing side-channels is to adhere to the constant-time policy. Modern out-of-order architectures take a central role in this line of research and balancing execution timings is not effective on these systems. The constant-time programming policy avoids secret-dependent control-flow altogether [4], [8], [16]–[18], [35], [41], [42], [46]. This policy is effective against the Nemesis attack, since there are no secret-dependent branches, only straight-line code. On high-end processors, where attackers can exploit advanced microarchitectural features, adhering to the constant-time policy is arguably the only reliable and future-proof defence, although recent work [15], [23] shows that classical constant-time programming is not effective either in presence of out-of-order and/or speculation. On the other hand, low-end architectures such as AVR, MSP430, and certain ARM and RISC-V cores have very different microarchitectural designs with deterministic timings which allow for more relaxed policies.

Automated approaches to detect and mitigate timing vulnerabilities have been proposed [10], [36], [44]. Language-based techniques that deal with side-channels typically focus on the detection of side-channels, rather than program hardening through code transformation. Furthermore, existing proposals for program transformations are often limited to simple, theoretical languages.

A compiler lowers high-level language abstractions into low-level processor instructions, enabling it to control how information may leak. Similar to how compilers optimise code, based on execution models of the target architecture and algorithms, e.g. for register allocation and instruction scheduling, it is possible to create architecture-specific leakage models to support the automatic detection and hardening of vulnerable code. Compiler-based approaches have been proposed at different abstractions levels [16]–[18], [20], [41], [46]. Mantel and Starostin [34] experimentally evaluate four source-to-source transformations for mitigating timing side-channel vulnerabilities: cross-copying [3], conditional assignment [35], transactional branching [9], and unification [30].

Our work presents a novel defence that mitigates timing side-channels and interrupt-latency attacks on lightweight embedded platforms, and that aims to reduce performance overheads on these platforms. In difference to all related work, we have implemented our defence in the backend of a compiler. We argue that timing leaks should be transformed out as late as possible in the compilation pipeline, preferably in the compiler backend. This renders the transformation source-language independent, reduces the risk of later compiler passes from breaking the protections, and allows for earlier passes to be implemented without security considerations. The LLVM X86CmovConversion pass, e.g., converts x86 `CMOV` instructions into branches when profitable, possibly breaking the security hardening by an earlier if-conversion [18].

6.2. Comparability of Experimental Results

Compile-time defences against low-level side-channel attacks are bound to be rather specific to an execution platform and a leakage model, but also to a compiler architecture, source languages and intermediate program representations, and even application domains. With a focus on mitigating a novel attack on embedded execution platforms, our work is no different. To the best of our knowledge, there are neither generic benchmarks nor related works that evaluate a defence designed to prevent information leakage through interrupt side-channels. While embedded execution platforms similar to AVR or MSP430 microcontrollers are rather wide-spread, software development for these platforms is typically done in low-level languages and many toolchains and compiler front-ends for higher-level languages do no support code

generation for embedded platforms. This makes it difficult to directly compare our transformation with existing work that typically focuses on desktop architectures. In the following paragraphs we focus on explaining why a comparison with relevant and recent related publications is currently not feasible. Developing a benchmark and evaluation approach for the security-enhancing compilation to embedded platforms would be interesting but goes far beyond the scope of this paper.

Most recently, Wu et al. [46] evaluated an LLVM-based program transformation that hardens cryptographic library code against both instruction and cache timing side-channels. The authors introduce a new compiler intrinsic, which we would have to implement for embedded target architectures, and the results of a comparison would be highly dependent on the quality of this implementation. Furthermore, implementation artifacts of [46] are not available in source code, requiring us to either re-implement the proposed algorithms or exchange the output of intermediate compilation stages across different versions of LLVM. We deem these options to be either not viable, or to yield unreliable results.

In [34], Mantel and Starostin experimentally evaluate transformations that remove timing side-channel vulnerabilities. The evaluation relies on four relatively simple benchmark programs, comparable with the ones we use. However, the programs in [34] are written in Java, compilation of which to the MSP430 is not supported. We do not expect a comparison of Java byte-code with native embedded code to yield reliable results as both runtime environments vary fundamentally in feature-richness.

The use of if-conversion in a compiler backend to eliminate secret-dependent timing is first proposed by Coppens et al. in [18], and evaluated based on three exponentiation functions and an RSA implementation from the OpenSSL library. We argue that these benchmarks are rather trivial in terms of control-flow complexity as they cover only a subset of the cases contained in our benchmark. The OpenSSL function is an interesting use case that we may consider in future work; still, a direct comparison is not possible since Coppens et al. target the x86 architecture and implementation artifacts of the program transformations are not available. The transformations in [18] are similar to what is proposed by Molnar et al. in [35]. Molnar et al. evaluate these transformations based on two implementations of cryptographic algorithms, RC5 and IDEA. Reproducibility and comparability is again not feasible due to the unavailability of these transformations in a recent compiler toolchain.

A code transformation that is based on the execution of "decoy paths", which are later merged through predicated store instructions, is proposed by Rane et al. in [41]. While the program transformations appears to be similar to [18], [35], Rane et al. implement their approach as a transformation of a program's Intermediate Representation (IR) in LLVM. A mean performance overhead of 16.1x is reported. While this overhead is worse than what we report in this paper, differences between the target architectures and attacker models do not warrant a direct comparison. The benchmarks from [41] are not publicly available.

A range of related publications, including e.g., [36] and [3] do not evaluate the performance and size of the post-transformation code at all.

6.3. Orthogonal Approaches

Recently, Busi et al. [14] propose a hardware-based countermeasure against interrupt-latency attacks for interruptible Sancus [37] enclaves. Their approach changes the fetch-decode-execute operation of the processor so as to pad out observable timing differences. In contrast to our work, this approach requires hardware modifications and is thus not applicable to off-the-shelf devices, including the ones already deployed in the field.

7. Conclusion

We present and evaluate an automated approach to harden embedded enclave programs at compile time against sensitive information leakage through a novel class of side-channel attacks. Our approach is based on a semantics-preserving program transformation to guarantee that branches in secret-dependent control flow each have the same number of instructions with the same per-instruction execution times. We implement and evaluate our transformation as a compiler pass that operates on low-level machine IR in LLVM, showing that interrupt latency vulnerabilities are eliminated. Our approach is applicable to programs for low-end processors with predictable instruction execution times; we evaluate a prototype of the compiler pass for the TI MSP430 microcontroller, and project the feasibility of implementing the defence for AVR and certain ARM-based processors. Our results indicate that smaller and more efficient code can be generated for these processors than by following the constant-time policy, while still securing against timing side-channels. This allows us to conclude that balancing branches can be preferable over constant-time programming in some situations. We anticipate that our approach will enable the hardening of embedded enclave programs beyond cryptographic functions, e.g., in critical control systems or IoT devices.

In future work we may extend our approach to provide protection against additional side-channel attacks, consider a range of possible optimisations in our algorithm and implementation, extend the comparison with the constant-time policy, and envisage an evaluation on an extended application scenario. Another future line of work could be a formalisation and correctness proof of the algorithm.

Acknowledgments

We thank Jo Van Bulck, Raoul Strackx and Job Noorman for helpful discussions about this work. We thank Jo Van Bulck and Raoul Strackx for comments on an early draft of this paper. We thank the anonymous reviewers for helpful comments that helped improving the paper. This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. This work was partially supported by a gift from Intel Corporation.

References

[1] "Botan: Crypto and TLS for Modern C++," https://github.com/randombit/botan/.

[2] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 312–320.

[3] J. Agat, "Transforming out timing leaks," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2000, pp. 40–53.

[4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.

[5] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[6] ARM, "Cortex-M23," https://developer.arm.com/ip-products/processors/cortex-m/cortex-m23.

[7] Atmel, "AVR Instruction Set Manual," 2016, https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.

[8] 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)*. IEEE, 2018, pp. 328–343.

[9] G. Barthe, T. Rezk, and M. Warnier, "Preventing timing leaks through transactional branching instructions," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 33–55, 2006.

[10] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 230–235.

[11] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 159–176.

[12] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Design Automation Conference (DAC '15)*. IEEE, 2015, pp. 1–6.

[13] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[14] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Provably secure isolation for interruptible enclaved execution on small microprocessors," in *33rd IEEE CSFW*. IEEE Computer Society, 2020.

[15] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 913–926.

[16] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.

[17] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: a DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 174–189.

[18] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 45–60.

[19] W. Daniels, D. Hughes, M. Ammar, B. Crispo, N. Matthys, and W. Joosen, "S μ v-the security microvisor: a virtualisation-based security middleware for the internet of things," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, 2017, pp. 36–42.

[20] F. Dewald, H. Mantel, and A. Weber, "AVR processors as a platform for language-based security," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 427–445.

[21] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *19th Annual Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[22] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[23] R. Guanciale, M. Balliu, and M. Dam, "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1853–1869.

[24] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–Bringing access-based cache attacks on AES to practice," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 490–505.

[25] T. Instruments, "MSP430 competitive benchmarking," 2005, http://www.mcuzone.com/work/DIMM144-CPU-MSP430/slaa205a.pdf.

[26] ——, "MSP430x1xx Family: User's Guide," 2006, https://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[27] ——, "MSP430-GCC-OPENSOURCE," 2020, https://www.ti.com/tool/MSP430-GCC-OPENSOURCE.

[28] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[29] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys '14*. ACM, 2014, p. 14 pages.

[30] B. Köpf and H. Mantel, "Transformational typing and unification for automatically correcting insecure programs," *International Journal of Information Security*, vol. 6, no. 2-3, pp. 107–131, 2007.

[31] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[32] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: A framework for architecting TEEs," *arXiv preprint arXiv:1907.10119*, 2019.

[33] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 557–574.

[34] H. Mantel and A. Starostin, "Transforming out timing leaks, more or less," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 447–467.

[35] 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 *International Conference on Information Security and Cryptology*. Springer, 2005, pp. 156–168.

[36] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 58–75.

[37] 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)*, vol. 20, pp. 7:1–7:33, 2017.

[38] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[39] C. Percival, "Cache missing for fun and profit," 2005.

[40] 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*, 2020, pp. 1–10.

[41] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[42] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1697–1702.

[43] 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*. ACM, 2018, pp. 178–195.

[44] C. Wang and P. Schaumont, "Security by compilation: an automated approach to comprehensive side-channel resistance," *ACM SIGLOG News*, vol. 4, no. 2, pp. 76–89, 2017.

[45] H. Winderix, "Security Enhanced LLVM," Master's thesis, KU Leuven, 2018, https://distrinet.cs.kuleuven.be/software/sancus/publications/winderix18thesis.pdf.

[46] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

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

[48] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.