Computer Networks 205 (2022) 108744

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

NemesisGuard: Mitigating interrupt latency side channel attacks with static binary rewriting

Majid Salehi [a,*], Gilles De Borger [a], Danny Hughes [a], Bruno Crispo [b]

[a] imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
[b] University of Trento, 38122 Trento, Italy

ARTICLE INFO

Keywords:
IoT security
Side channel attacks
Static binary instrumentation

ABSTRACT

Internet of Things (IoT) is becoming integrated into nearly every aspect of our modern life. Indeed, exploitation of such devices can directly lead to physical consequences in the real world. Previous work has shown that IoT devices can be compromised by exploits in lower software layers such as the Operating System (OS). Embedded Trusted Execution Environments (TEEs) provide a small Trusted Computing Base (TCB) to protect sensitive codes and data in such devices. TEEs assume a strong threat model where even a privileged attacker (e.g. OS) cannot compromise the confidentiality and integrity of the execution. Nevertheless, it has been shown that side channel attacks make it challenging to keep secrets during application execution.

Interrupt latency side channel attacks (a.k.a. Nemesis) are a novel type of timing attacks that target embedded TEEs and extract application secrets from them. Nemesis attacks exploit the CPU's interrupt mechanism to reveal microarchitectural instruction timings from embedded TEEs. Specifically, the attacker measures the latency of a precisely timed interrupt to differentiate between secret-dependent branches. In this paper, we present NemesisGuard, the first mitigation mechanism against such side channel attacks that does not require a modified compiler or hardware and can protect COTS binaries without access to source code. NemesisGuard applies a novel static binary instrumentation technique to balance secret-dependent branches in IoT application binaries. Evaluation of NemesisGuard shows that it mitigates Nemesis side channel attacks effectively and efficiently.

1. Introduction

Internet of Things (IoT) devices are increasingly used for applications that can have devastating and immediate consequences if compromised—including implantable medical devices, automotive, and industrial control systems. The number of IoT devices is expected to increase to around 19 billion worldwide by 2022 [1]. However, these devices increasingly face threats from exploits in lower software layers such as the Operating System (OS). In order to protect security-sensitive codes and secrets from these threats, there is an increasing interest in hardware implementations of primitives for trusted computing such as ARM TrustZone [2], TrustLite [3], or Sancus [4,5] Trusted Execution Environments (TEEs).

Indeed, TEEs guarantee the confidentiality and integrity of secrets and applications without trusting other software components such as the OS. In order to use TEEs, users partition their applications into secure containers, called enclaves, and encapsulate sensitive application code and data in them. Created enclaves are completely isolated from other applications and even from the OS.

Nevertheless, side channel attacks make it challenging to keep secrets during application execution. Side-channel attacks have been used to infer confidential information from nonfunctional characteristics of computations such as time [6], memory [7], or power consumption [8]. Particularly popular are timing side-channel attacks that intend to recover secrets through measuring the execution time behavior of the algorithm implementation. Indeed, the key idea of these attacks is based on the assumption that the execution time of an algorithm implementation depends on the value of secret information. Several methods [9–12] are proposed to protect against timing side channel attacks; most of them modify the application code by inserting some compensation codes to the secret-dependent paths. In other words, these methods harden application's control-flow paths to ensure that their execution time is never secret-dependent.

Recently, Van Bulck et al. [13] introduced the first remotely exploitable microarchitectural side channel attack, called Nemesis, that is applicable to embedded TEEs. Nemesis attack exploits the timing

* Corresponding author.
E-mail addresses: majid.salehi@cs.kuleuven.be (M. Salehi), gilles.deborger@student.kuleuven.be (G.D. Borger), danny.hughes@cs.kuleuven.be (D. Hughes), bruno.crispo@unitn.it (B. Crispo).

https://doi.org/10.1016/j.comnet.2021.108744
Received 31 May 2021; Received in revised form 17 December 2021; Accepted 21 December 2021
Available online 11 January 2022

differences in the rudimentary fetch-decode-execute operation of CPUs. The attacker measures the latency of a precisely timed interrupt (i.e. interrupt latency) to differentiate between secret-dependent branches. Compared to other timing side channel attacks, Nemesis attack is able to leak instruction-granular timings from enclaved execution environments. Consequently, the traditional mitigation methods for timing attacks are not applicable in the case of Nemesis attacks.

To mitigate Nemesis attacks, Busi et al. [14,15] designed and developed a novel hardware-based solution that provides an enclave-enabled microprocessor with secure interruptibility of the enclaves. Specifically, they modify the fetch-decode-execute operation of CPUs by padding observable timing differences. However, hardware-based solutions are not practical for the IoT devices that are already deployed in the field. Winderix et al. [16] proposed a compiler-based approach that hardens MSP430 applications with compensation codes against Nemesis attacks. Compiler-based approaches attempt to lift the code to an Intermediate Representation (IR) and then apply the hardening procedure on it. Unfortunately, these approaches are ineffective for Commercial Off-The-Shelf (COTS) binaries since they require to correctly recover type information from binary files in order to lift the code to IR for instrumenting and hardening, a process that remains an open problem [17, 18].

In this paper, we propose NemesisGuard, a novel approach that statically instruments embedded COTS binaries to automatically mitigate Nemesis side channel attacks. NemesisGuard provides a static binary instrumentation method and uses it for instrumenting secret-dependent branches without any need to lift assembly to a higher-level intermediate language. NemesisGuard allows to insert compensation codes based on a novel and efficient algorithm for protecting against leaking secrets from embedded TEEs using the interrupt latency channel. In summary, the paper makes following contributions:

- We propose NemesisGuard, the first static-binary-rewriting based technique to mitigate interrupt latency side channel (a.k.a. Nemesis) attacks in embedded binaries. In contrast to the state-of-the-art, our technique does not require a modified hardware and also avoids lifting assembly to an intermediate language, a process which is known to be error-prone [17].
- We have implemented the technique in a full-featured framework for the ARM architecture which is one of the most widely used IoT device architectures. To foster further research, we make our framework prototype available open source.
- We experimentally evaluated the effectiveness of the proposed framework using 11 real-world IoT application binaries and demonstrated that it allows mitigation of interrupt latency side channel attacks. The results also show the efficiency of instrumented binaries in practice.

2. Background and motivation

In this section, we present a brief overview of TEEs for protecting secrets. Furthermore, we describe timing side channel attacks targeting embedded TEEs and code instrumentation as an approach to mitigate them. We also discuss the limitations of the proposed instrumentation approaches to mitigate interrupt latency side channel attacks that motivate the need to extend and refine binary instrumentation for such attacks.

2.1. Trusted execution environments

There have been several studies [2–5,19–21] on trusted execution on untrusted OS. Most of them rely on a Trusted Computing Base (TCB) that protects a shielded memory region against both confidentiality and integrity attacks. Indeed, that memory region is safeguarded against external software running on the system regardless of its privilege levels. This security mechanism provides developers with an unprecedented capability to develop security-critical applications and encapsulate them inside an enclave, achieving strong security guarantees in terms of confidentiality and integrity even under the assumption of a malicious OS.

TEEs are typically developed based on an execution model that allows untrusted code to interrupt trusted enclaves. Therefore, since advanced performance-improving techniques that cause timing variations such as caches, branch predictors, and out-of-order pipelines are not present, low-end embedded TEEs are based on CPUs with predictable instruction execution times. These properties of embedded TEEs, as demonstrated by Van Bulck et al. [22], results in a noiseless type of microarchitectural timing side-channels that rely on interrupt latency.

2.2. Timing side channel attacks

Different side channels, like performance, power consumption, timing, etc. can be used by the attacker to derive useful information about the victim (e.g., cryptographic keys) and mount the attack. Timing side channels [23–27] are usually exploited to measure the execution time of an algorithm implementation; this can be done directly in the device itself or by interacting remotely with the code over a network [22].

To describe timing attacks clearly, Fig. 1(a) shows an enclave application [14], that compares user-provided password in *r6*, with a password stored in the enclave, and stores the value in *r2* into the enclave location at *store_adrs* if the user-provided password is correct. It is clear that the execution time of two possible control flow paths for the if-branch in this code is unbalanced (3 cycles for *if* and 4 cycles for *else*). Therefore, an attacker can use a cycle accurate timer for measuring the enclave start-to-end execution time by setting the timer right before entering to the enclave. By doing so, the attacker will be able to apply a brute-force attack with the aim of inferring the correct password [28].

Current countermeasures [29,30] against these types of attacks attempt to level the timing behavior of both control flow paths by appending some dummy instructions to the code. Fig. 1(b) shows the balanced code which is protected against such kind of attacks (4 cycles for *if* and 4 cycles for *else*). However, recently, security researchers in [13] have shown that even after balancing the branch and removing the dependencies between the execution times of the paths and password, attackers can still extract the correct secret by a novel interrupt-based side channel attack called Nemesis. In the following, we describe the Nemesis attack and its variants in details.

2.3. Nemesis-type interrupt timing attacks

Nemesis attack exploits a subtle timing channel in the CPU's rudimentary interrupt mechanism. There are three variants of Nemesis attack which are originally proposed in [13,14]:

Basic Nemesis Attack: Basic Nemesis attack assumes a multi-cycle instruction set architecture, where interrupt requests are only served after completing the current executing instruction. This is an assumption which is valid for major embedded instruction set architectures. In such architectures, interrupt handling mechanism introduces a timing difference between arrival of the hardware interrupt request and execution of the first instruction in the interrupt service routine (i.e. interrupt latency).

Indeed, interrupt latency is increased by the number of cycles remained to execute if there is an interrupt request during the execution of a multi-cycle instruction. Therefore, the attacker can request an interrupt at a specific time during the execution of a multi-cycle instruction and determine the duration of the interrupted instruction by observing the timestamp placed on interrupt service routine entry, enabling him/her to differentiate between secret-dependent branches.

For example, in Fig. 1(b), the attacker can schedule an interrupt to arrive within the first clock cycle after the conditional jump instruction at line 6 (*beq access_ok*). By doing so, either the 3 cycle *b end_if*

```
enclave_entry
    mov r1, #store_adrs
    mov r4, #pwd_adrs
    ldr  r4, [r4]   ;load secret enclave password in r4
    cmp  r4, r6     ;compare user vs. enclave password
    beq  access_ok
access_fail         ;password fail
    b end_if        ;3 cycles
access_ok           ;password ok: store user value

    str  r2, [r1]   ;2 cycles
end_if              ;clear secret enclave password
    sub  r4, r4     ;1 cycle
enclave_exit
```

(a) Vulnerable against Start-to-end timing attacks.

```
enclave_entry
    mov r1, #store_adrs
    mov r4, #pwd_adrs
    ldr  r4, [r4]   ;load secret enclave password in r4
    cmp  r4, r6     ;compare user vs. enclave password
    beq  access_ok
access_fail         ;password fail
    b end_if        ;3 cycles
access_ok           ;password ok: store user value
    nop             ;1 cycle
    str  r2, [r1]   ;2 cycles
end_if              ;clear secret enclave password
    sub  r4, r4     ;1 cycle
enclave_exit
```

(b) Mitigated against Start-to-end timing attacks but vulnerable against Nemesis attacks.

```
enclave_entry
    mov r1, #store_adrs
    mov r4, #pwd_adrs
    ldr  r4, [r4]             ;load secret enclave password in r4
    cmp  r4, r6               ;compare user vs. enclave password
    beq  access_ok
access_fail                   ;password fail
    str r0, [sp, #-4]         ;2 cycles
    b end_if                  ;3 cycles
access_ok                     ;password ok: store user value
    str  r2, [r1]             ;2 cycles
    b temp_label              ;3 cycle
temp_label
end_if                        ;clear secret enclave password
    sub  r4, r4               ;1 cycle
enclave_exit
```

(c) Mitigated against Start-to-end timing and Nemesis attacks.

Fig. 1. Assembly code of a running enclave example that first loads the enclave password (stored in pwd_adrs) and compares it with the user provided password (stored in r6 register). If the user password is correct, it will store the user provided input in store_adrs.

instruction or 1 cycle *nop* will be interrupted. Indeed, based on the branch condition, there is an interrupt latency difference of 2 cycles. The attacker can measure the interrupt latency and determine the correctness of individual bytes and finally extract the correct password value. Basic Nemesis attacks can be mitigated by modifying the interrupt handling mechanism to have a constant time interrupt latency regardless of the interrupted instruction's execution time. For instance, if we use a hardware patch for the interrupt handling mechanism to always dispatch interrupts in the worst-case interrupt response time, the interrupt latency would be same for every instruction, causing the application to be protected against basic Nemesis attack. However, they are still vulnerable to the advanced types of Nemesis attacks, which are described in what follows.

Resume-to-end Nemesis Attack: In case the processor has a constant time interrupt latency, the attacker can request an interrupt to arrive within the first clock cycle after the conditional jump. After serving the interrupt, the attacker will let the enclave to resume and measure the time the enclave takes to complete the task without any additional interrupts. Although there is no difference in the interrupt latency for the *if* and *else* paths, the execution time for completing the enclave task after serving interrupt is 3 cycles for the *if* path and 1 cycle for the *else* path. As a result, the attacker can determine which path is executed for the current input in order to extract the password stored in the enclave.

Interrupt-counting Nemesis Attack: As another variant of Nemesis attacks (i.e. interrupt counting), the attacker can count the total number of the times that the enclave execution can be interrupted. The only assumption of this attack is a multi-cycle instruction set, thus it is also applicable on processors with a constant-time interrupt latency. Indeed, in such processors, due to the fact that interrupts are handled on instruction boundaries, it would be possible for the attackers to count the number of instructions in each branch and determine the branch that is being executed. For instance, in Fig. 1(b), the attacker can request new interrupts one cycle after resuming from the previous one. Specifically, the *if* path can be interrupted 3 times compared to the *else* that can be interrupted 2 time. Therefore, the attacker can determine the executing path to infer the correct password bytes by scheduling single-stepping interrupts.

3. Threat model

The attacker's goal is to extract secret information regarding the internal state of an enclaved application. In this regard, TEEs have been specifically proposed to protect sensitive computations on an untrusted attacker-owned platform, both in an untrustworthy cloud environment [31,32] and to enforce enterprise right management on consumer hardware [22,33,34]. Precisely, the aim of the attacker is to use unprotected parts of the system in order to derive application secrets from a vulnerable enclave by scheduling interruption of the enclave. We assume an attacker with an access to the compiled or source code of the victim application. We also assume that the attacker has a full control over the unprotected parts of the system by TEE such as OS; the attacker is able to configure hardware devices (e.g. timer) and load malicious modules on the system. Indeed, protection against these types of side channel attacks is much harder than process isolation mechanisms which consider the OS trusted.

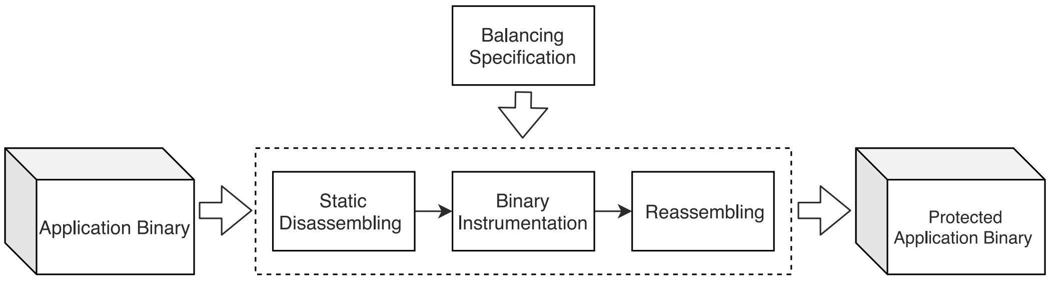


Fig. 2. Pipeline of NemesisGuard.

For instance, a realistic scenario for these properties could be an embedded device that the attacker and the victim connect to over a network. Indeed, the victim interacts with the embedded device to execute a critical application in a TEE. The attacker could infer the state of the victim program by observing the network and evaluating the critical application's start-to-end execution time, communication patterns, or by deploying a spy module and dynamically manipulating timers and interrupts to launch a Nemesis attack.

Since we protect against timing side channel attacks, physical side channel attacks targeting power consumption and electromagnetic radiation are out of scope and orthogonal to our research. Furthermore, side channel attacks targeting advanced microarchitectural CPU features such as paging, caching, branch prediction, or out-of-order execution are out of scope since these CPU features are not available in embedded devices [22]. Our threat model is consistent with prior research on Nemesis side channel attacks and protection mechanisms.

In this work, we focus on ARM architecture as the most widely used architecture for embedded systems [35]. Thus, we consider TEEs designed for this architecture like ARM TrustZone [2]. Nevertheless, the conceptual approach can work also with other architectures and TEEs with extra engineering efforts (see Section 8). Furthermore, we do not consider any self-modifying or packed binaries (such as self-extracting compressed application).

4. NemesisGuard

Fig. 2 shows a high-level overview of the NemesisGuard framework. NemesisGuard takes as input a binary and produces an instrumented version protected against Nemesis side channel attacks, through binary instrumentor and balancing modules. In other words, balancing module leverages binary instrumentor module to immune binary application against all variants of Nemesis side channel attacks by aligning the execution time of corresponding instructions in secret-dependent branches. The modified binary can be seen as split in two portions: the instructions considered not sensitive against attacks (i.e. placed in non secret-dependent branches), which are left unmodified by the process, and the sensitive ones (i.e. placed in secret-dependent branches), which are not emitted. In this section, we describe the most important and challenging aspects of these modules.

4.1. Static binary rewriting

This module carries out the central task, instrumenting the binary to harden secret-dependent branches. This is done in three main steps: the static disassembling, the binary instrumentation, and the reassembling.

Static Disassembling. The first step of our framework pipeline disassembles binary file's code section using a linear sweep approach. Specifically, this approach parses the code section from the beginning to the end and decodes all encountered bytes as instructions. As pointed out by Andriesse et al. [36], linear sweep outperforms other disassembly approaches such as recursive traversal [37–39] and achieves close to 100% accuracy for instruction disassembly from compiler-generated binaries.

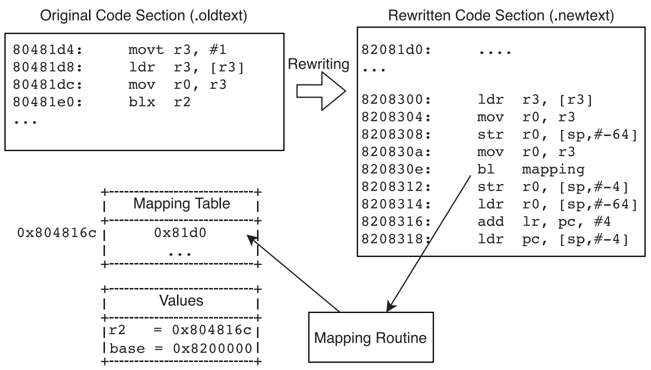
When given an application disassembly, NemesisGuard will build its best-effort Control Flow Graph (CFG) starting from the main function by identifying and adding edges for direct branches. Indeed, NemesisGuard does not require precise recovery of the CFG as indirect calls and jumps are handled by the instrumentor module at the runtime when the exact target address of them are known, rather than heavyweight static analysis. Thus, we generate an over-approximated CFG which considers the possible target sets for indirect branches. As it is described in Section 4.2, this over-approximated CFG can be used by NemesisGuard to balance all secret dependent branches, even the ones that are not actually taken at the runtime. Specifically, we can tolerate false positives (branches belong to the target sets according to the analysis, but are not actually taken at the runtime) but not false negatives.

Binary Instrumentation. At the second step, our framework takes the disassembled application as an input and inserts compensation instructions to the secret-dependent branches. Nevertheless, because of the lack of source-level information, binary instrumentation introduces challenges that are not present when editing source code or compiler-generated assembly files. More specifically, when instructions are inserted at the source code level, the compiler will rearrange code and data in memory and handle references between them. However, at the linking time, the compiler discards the symbol and relocation information and hard-codes labels to specific addresses. Therefore, inserting instructions breaks all references in the binary.

Our framework addresses these challenges by duplicating the code section in a new section (called .newtext) that contains the rewritten executable code. It also preserves the old copy of code section (called .oldtext) serving as a read-only data segment. By preserving .oldtext and data sections at their original addresses intact, data pointers in the rewritten code section .newtext continue to behave correctly.

Then, our framework adjusts all the target addresses of code pointers while rewriting them in the .newtext section in order to ensure that they point to their targeted locations. Precisely, direct branches instructions are statically rewritten to reference their new target addresses. Indirect branch instructions have multiple possible targets and their exact target addresses are only known at runtime. Unlike many prior efforts [40–42], we observe that although it is challenging to statically determine exact target addresses of indirect branches, we can instead apply an efficient dynamic lookup at runtime.

Indeed, our framework adds a level of indirection to solve the indirect branches problem, our static phase translates all indirect branches from .oldtext section into the alternative instructions in the .newtext section that dynamically detects and re-points old pointers to new target addresses at runtime. Specifically, our framework generates a mapping table containing each possible target address in the .oldtext section mapped to the corresponding address in the .newtext section. For instance, as depicted in Fig. 3, the framework rewrites original instructions from the .oldtext section along new inserted instructions in the .newtext section with new base address 0x8200000. It translates every indirect branch (*blx r2*) into the *mov* and direct call instructions. In fact, the *mov* instruction moves the original target address of indirect branch (0x804816c) into register r0 and the direct call (*bl mapping*) goes to the mapping routine to search for the offset corresponding to


Original Code Section (.oldtext)
Rewriting
Rewritten Code Section (.newtext)
Mapping Table
0x804816c
0x81d0
...
Values
Mapping Routine

```
80481d4:     movt r3, #1
80481d8:     ldr  r3, [r3]
80481dc:     mov  r0, r3
80481e0:     blx  r2
...
```

```
82081d0:     ....
...

8208300:     ldr  r3, [r3]
8208304:     mov  r0, r3
8208308:     str  r0, [sp,#-64]
820830a:     mov  r0, r3
820830e:     bl   mapping
8208312:     str  r0, [sp,#-4]
8208314:     ldr  r0, [sp,#-64]
8208316:     add  lr, pc, #4
8208318:     ldr  pc, [sp,#-4]
```

```
r2   = 0x804816c
base = 0x8200000
```

Fig. 3. The binary instrumentor module redirects all indirect branch instructions (e.g., blx r2) to the mapping routine which looks for new offset (0x81d0) corresponding to the old target address (0x804816c) in the mapping table.

the old target in the mapping table (0x81d0). If the search succeeds, mapping routine returns new translated target address (0x8200000 + 0x81d0 = 0x82081d0) for jumping (*ldr pc, [sp, #-4]*) to it accordingly.

Reassembling. In the reassembling step, our framework takes the instrumented assembly code and reassembles it as a working binary using off-the-shelf assemblers.

4.2. Balancing specification

We already discussed (in 2.2) that it is insufficient for security to naively add instructions in order to balance branch's total execution time. The balancing specification determines which exact instructions should be instrumented by NemesisGuard. It tries to add compensation codes to the secret-dependent branches such that the execution times of the instructions that have a same distance from a common secret-dependent branch instruction always be equal to each other. It is worth mentioning that inserted compensation codes should preserve the original application characteristics such as control flow and register and memory access patterns.

For instance, in Fig. 1(C), NemesisGuard adds two compensation instructions to the secret-dependent branch (i.e. *beq access_ok*): *str r0, [sp, #-4]* instruction[1] in *access_fail* branch and *b temp_label* in *access_ok* branch. By doing so, the execution time of every corresponding instructions in *access_fail* and *access_ok* are aligned; it is not possible for the attacker to extract secrets by applying Nemesis attacks.

Secret-dependent branches can be identified automatically through the methodology proposed in SelectiveTaint [43]. Indeed, SelectiveTaint proposes an efficient selective taint analysis framework for x86 binary executable. SelectiveTaint scans taint sources of interest in the binary, utilizes value set analysis (VSA) to decide whether an instruction operand will be involved in taint analysis, and then selectively taints the instructions of interest. We refer readers interested in a detailed explanation on the different aspects of SelectiveTaint to its original paper since the aim of this work is not to advance the state-of-the-art in taint analysis, and we consider this orthogonal research.

We apply a same static taint analysis approach upon our static binary instrumentor module by assigning a security level to function arguments and tracking them as they propagate along the application execution path. Finally, branch instructions with operands that are assigned sensitive value, i.e. branches dependent on sensitive data, can be considered sensitive, and thus subject to binary instrumentation (i.e. balancing).

[1] We are storing r0 at sp-4 memory address in order to avoid overwriting the stack values.

After identifying secret-dependent branches, we aim to balance them by satisfying Nemesis-sensitive property. Pouyanrad et al. [44] introduces the Nemesis-sensitive property to formally guarantee the absence of all types of Nemesis vulnerabilities in an application:

Definition 1. Considering application A's extracted CFG, A satisfies Nemesis-sensitive property if and only if:

$$\forall IP^i \in BB^{then}(IP) : \forall IP^j \in BB^{else}(IP) \; such$$
$$that \; i = j : (s_{IP^i} \xrightarrow{t} s_{IP^{(i+1)}}) \wedge (s_{IP^j} \xrightarrow{t'} s_{IP^{(j+1)}}) \Leftrightarrow t = t'$$

where two functions $\mathrm{BB}^{\mathrm{then}}(\mathrm{IP})$ and $\mathrm{BB}^{\mathrm{else}}(\mathrm{IP})$ capture the set of instruction points (IP) belonging to the if and else basic blocks respectively. The number of instruction points captured by $\mathrm{BB}^{\mathrm{then}}(\mathrm{IP})$ and $\mathrm{BB}^{\mathrm{else}}(\mathrm{IP})$ should be same. IP^{i} is an instruction points at depth i of its own basic block. Symbol t is defined as the time of transition from one instruction point to another one.

Definition 2. An instruction point IP is said to be at depth n in basic block BB if there is a path from the entry point of the BB to IP with length n.

Definition 3. Basic block BB is said to be at level l in CFG if there is a path from the root of the CFG to BB with length l. Indeed, since it is possible that there are multiple paths with different lengths to a node, BB can be at multiple levels.

As a result, NemesisGuard instruments instructions in secret-dependent branches with the aim of satisfying Nemesis-sensitive property. By doing so, it would not be possible for an attacker to apply Nemesis attacks and extract secrets from TEE enclaves by measuring interrupt latency. In what follows, we describe details of balancing algorithm for satisfying Nemesis-sensitive property.

Balancing Algorithm. Algorithm 1 depicts NemesisGuard's balancing algorithm. In this algorithm, the nodes (i.e. basic blocks) of the secret-dependent branch's CFG are balanced in a level-wise manner. Our balancing algorithm assumes that all paths to a given node have the same length. In other words, every nodes in the secret-dependent branch's CFG is at no more than one level. In the case that there are some nodes with more than one level value, NemesisGaurd first follows the equalizing procedure which will be described later to equalize branches. Thereafter, as shown in *AlignCFG* procedure in Algorithm 1, NemesisGuard iterates over all the levels of the graph and balances the nodes found at that level. In fact, additional instructions are inserted into nodes to ensure that corresponding instructions in a same depth have the same latency.

The core of the balancing process consists of repeatedly selecting a reference instruction from one of the nodes in a specific level and inserting instructions into the other nodes in that level such that the latencies match (i.e. *AlignNodes* procedure in Algorithm 1). The node from which the reference instruction is selected is called the reference node. The reference node can change throughout the algorithm. Because an instruction is potentially added to each node that is not the reference node, the reference node needs to have at least as many instructions as the node with the largest number of instructions. This ensure that at some point all nodes have the same number of instructions.

Specifically, let n_{max} be the number of instructions in the longest node. The set of candidate nodes then consists of all nodes that have n_{max} instructions. Then, by applying the algorithm shown in function *SelectReferenceNode* in Algorithm 1, the reference node is selected from this set of candidates. An index variable is used to keep track of the position of the reference instruction. This variable is initially zero and is incremented every iteration. Any instructions that have an index smaller than this variable are balanced. The reference instruction is selected by first selecting the reference node and then selecting the instruction in this node that has an index equal to the index variable. However, reference instruction cannot be a branching instruction, unless during the very last iteration of the algorithm. This is due to the

Algorithm 1: Balancing Secret-dependent Branches

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

fact that inserting branching instructions into the node, if it is not the last instruction, will miss the execution of following instructions in that node, corrupting the normal operations of the node.

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

Equalizing Branches. Considering the CFG of secret-dependent branch, there are three cases that the aforementioned algorithm cannot be applied on. The first such structure occurs when the secret-dependent branch contains some sequences of instructions that are only executed if a condition is true, e.g. Fig. 4(A). In this structure, there will be two different paths with different lengths to one node. One path will contain the node that corresponds to the conditional instructions, while the other path will not contain this node. Since the paths have some overlapping nodes and the paths have different lengths, it is impossible to balance their latencies by inserting compensation instructions. In fact, inserting instructions in one of the paths will also insert instructions into the other path. If these paths start at a secret-dependent node, it is therefore impossible to ensure that the Nemesis-sensitive property holds.

The second challenging structure occurs when one of the branches is shorter than the other one, as shown in Fig. 4(A). In such cases there will be some nodes in one branch that have no corresponding nodes in the other branch, making it infeasible to balance them. The third challenging case includes loops in the secret-dependent branch, e.g. Fig. 5. Any node in a loop will be at infinite number of levels since there are infinite number of paths to it. The Nemesis-sensitive property entails that it is challenging for a secret-dependent branch to satisfy the property if one of these three structures occurs in its branches.

To remedy the first two challenging cases, NemesisGuard first equalizes all path lengths and branches before starting the above balancing algorithm. NemesisGuard inserts empty nodes into the graph to ensure that all paths to any given node are always with the same length (See Fig. 4(B)). Furthermore, in order to handle loops in a CFG, NemesisGuard first unwinds the graph, removing any cycles present. In particular, for a given loop, let the first node in the loop be the node closest to the root, and the last node the node furthest from the root. A new node is created identical to the first node in the loop. The edge that connects the last and the first nodes is removed and replaced by a new edge from the last node to the newly created copy of the first node (See Fig. 5). After the graph is balanced, the same procedure is executed in reverse in order to recreate the cycle. During balancing, any operation performed on the original node is also performed on the copy to ensure they are identical at all times.

Constructing NOP Instructions. For each latency class, a NOP instruction template has been determined. The instruction can be automatically inserted into the application as-is if it has no effect on the application state, i.e. it does not modify the original control flow, register values, and memory access patterns. Indeed, binary-only tools, unlike compiler-based ones, do not have the advantage of relying on virtual registers and letting the compiler allocate physical registers; instead, they must choose their own physical registers. Furthermore, the instrumentation process should not corrupt the application state, such as registers because this could cause unforeseen consequences, such as crashes or inconsistencies that are difficult to debug. If the

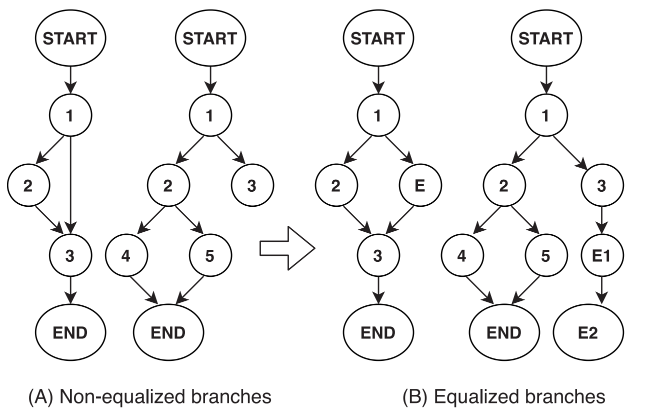


Fig. 4. NemesisGuard equalizes branches before applying balancing procedure—Left: two different cases of non-equalized CFGs that cannot be balanced. Right: Equalized instances of the left CFGs that are generated by inserting empty nodes.

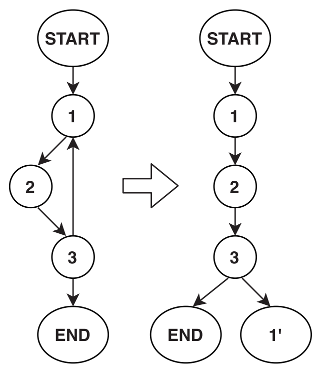


Fig. 5. Unwinding loops procedure.

instruction needs to modify a register, the algorithm selects a registers that can safely be used. This needs to be a register that is not in use at the time of execution of the instruction.

The function is statically analyzed to determine which registers are free to use for this purpose. Saving all state before entering instrumented code and restoring the saved state before departing is the safest option. This is, however, too expensive. Therefore, NemesisGuard applies an intra-function liveness analysis to discover all registers that are live (in-use) at instrumentation locations to save overhead from preserving application state. Specifically, before allocating live registers, NemesisGuard guarantees that non-live (free) registers are allocated first. Any allocated live register is also automatically saved and restored before and after instrumentation.

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

If a register of the first type exists then it can be used as the operand of the NOP instruction and the resulting instruction can be inserted as-is into the node. If no such registers exists, a free register of the second type is selected, and additional instructions are inserted into the application to ensure that the original value of the register is not lost. In the root of the CFG, additional instructions are inserted to push the register value onto the stack, while in every leaf, instructions are inserted that pop the value from the stack. If there are no free registers available, any register is arbitrarily selected. Then, 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.

5. Implementation

We have implemented a proof-of-concept of NemesisGuard for the ARMv7-M architecture [45], which covers a large share of microcontrollers (i.e., Cortex-M3/4/7) for embedded platforms [35]. The following outlines the technical details of the implementation based on the design described in the previous section.

5.1. Static binary instrumentation module

We have implemented the static binary instrumentation module on top of Capstone disassembler framework [46], spanning 1950 SLOC in Python programming language. This module leveraged pyelftools [47] open source framework to parse the ELF data structures. It also utilizes LIEF framework [48] for editing the header of application ELF file and creating a new code segment containing the .newtext section and mapping routine. It also used pwntools [49], an open-source binary analysis framework, as the platform for reassembling the instructions.

Table 1
IoT applications, their targeted MCUs, their used peripherals, and NemesisGuard effectiveness in satisfying Nemesis-sensitive property verified by our static analysis tool.

IoT Application	MCU	Peripherals	NemesisGuard
Audio-Playback	STMF479I-Eval	Clock, GPIO, USB, I2C	✓
LCD_Display	STMF479I-Eval	Clock, GPIO, SD-CARD, DSI	✓
LCD_Animate	STMF479I-Eval	Clock, GPIO, SD-CARD, DSI	✓
FatFs_uSD	STMF407G	Clock, GPIO, SD-CARD	✓
TCP_echo_Client	STMF479I-Eval	Ethernet, Clock, GPIO, EXTI	✓
TCP_echo_Server	STMF479I-Eval	Ethernet, Clock	✓
UDP_echo_Client	STMF479I-Eval	Ethernet, Clock, GPIO, EXTI	✓
UDP_echo_Server	STMF479I-Eval	Ethernet, Clock	✓
Camera-USB	STMF479I-Eval	Clock, GPIO, USB, DSI	✓
mbed-TLS	STMF401RE	Ethernet, Clock, GPIO, EXTI, DSI	✓
PLC	STMF479I-Eval	Clock, Timer, WiFI, UART, SPI	✓

5.2. Balancing module

We have implemented our balancing module on top of the binary instrumentor module with 870 SLOC in the Python language. Implementing binary-level balancing module on instrumentor module requires CFG recovery and instrumenting secret-dependent branches. The CFG is recovered as a part of NemesisGuard's disassembling procedure using angr [50,51]. After extracting CFG and determining secret-dependent branches based on the policy described in Section 4.2, we store them in the balancing specification file. Then, the binary instrumentor interprets the balancing specification file in order to balance all these branches with compensation codes.

6. Evaluation

This section presents the evaluation of various aspects of NemesisGuard. First, we demonstrate the effectiveness of NemesisGuard for mitigating Nemesis side channel attacks in IoT binaries in 6.2. Second, we evaluate the correctness of the instrumentation method against our dataset of IoT binaries in 6.3. Finally, we measured the runtime and space overhead of our instrumented and mitigated binaries in 6.4.

6.1. Experiment setup

As shown in Table 1, we collected 11 IoT applications for the experiments based on the following criteria: (1) The application in the experiment must be realistic, full-fledged, and deployed in market devices. (2) The application must be diversified in functionality.

(3) Applications in the experiments must cover the use of various peripherals such as Ethernet, camera, LCD display, serial port, SD card, and microphone to represent realistic interactions of IoT devices. Our collected applications collectively cover ARM Cortex-M3 and Cortex-M4 microcontrollers. These applications are provided with the development boards and written by STMicroelectronics [52].

In what follows, we provide a brief description of each application. Audio-Playback is an application developed for playing audio files by reading data from USB device and sending it to the audio codec. LCD_Display is an application for reading bitmap pictures from an SD card and displaying them on the LCD display. LCD_Animate creates the effect of animation by displaying multiple layers of bitmap images. In order to create animated pictures, the application displays an images sequence with a determined frequency on the LCD. Camera_USB application leverages the camera module to display pictures in a continuous mode on LCD and save them in a USB device. FatFs-uSD application implements a FAT file system on an SD card. TCP/UDP-Echo-Client/Server are four applications for running TCP/UDP echo client/server applications over Ethernet based on LwIP, which is a popular TCP/IP stack for IoT devices. mbed-TLS is an SSL client application that implements mbedTLS crypto library and LwIP TCP/IP stack on IoT devices. PLC (Programmable Logic Controller) is a family of IoT devices that has been designed and adapted for the control of manufacturing processes such as robotic devices. Our PLC application implements a PLC that executes uploaded ladder logic programs. The ladder logic program is uploaded to the microcontroller from an Android application via WiFi (ladder logic is a common PLC programming language).

As described in Section 4.2, it is developer's task to specify which function parameters are secret in the application. Then, our taint analysis tool can utilize those information to determine secret-dependent branches. However, in this section, we randomly chose 10 branches as secret-dependent branches in each IoT application described above for conducting experiments. Our experiments are performed on STM32-Nucleo F401RE [53], STM32F479I-Eval [54], and STM32F4Discovery [55] development boards featuring an ARM Cortex-M4 CPU and STM32-Nucleo L152RE [56] featuring an ARM Cortex-M3 CPU.

6.2. Effectiveness

We developed a static analysis tool to verify, for a given application, whether or not the application satisfies the Nemesis-sensitive property as specified in Section 4.2. We developed this tool inspired by SCF^{MSP} [44]. SCF^{MSP} defines a formal operational semantics for MSP430 instructions set that reflects execution time and information flow; it is then used as basis for proposing a security type system. The proposed security type system is leveraged to statically analyze MSP430 assembly codes to detect the possibility of information leakage through Nemesis side channel attacks. Inspired by SCF^{MSP}, we developed our static analysis tool for ARMv7-M architecture and then used that to verify the effectiveness of our approach in mitigating Nemesis attacks.

We instrumented the selected secret-dependent branches in application binaries in our dataset using the NemesisGuard balancing module and then applied our static analysis tool on instrumented binaries. As shown in Table 1, the static analysis tool verifies that Nemesis sensitive property holds for all instrumented binaries and NemesisGuard mitigates all Nemesis side channels without the need for a modified compiler or hardware as required by the state-of-the-art solutions [14–16].

6.3. Feasibility and correctness

We performed an evaluation on the feasibility and correctness of the static binary instrumentation module and its applicability to large real-world IoT applications by comparing the output of the original and instrumented binaries. Specifically, we executed the instrumented and original applications with the test suite shipped with the original one

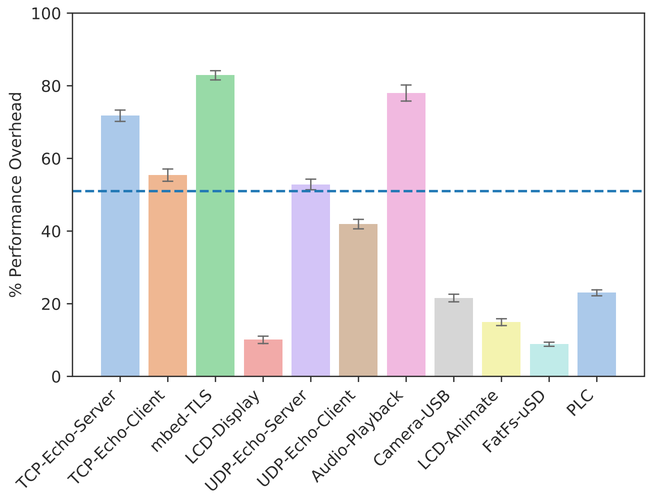


Fig. 6. Performance impact of NemesisGuard over the IoT application binaries.

to verify that all instrumented applications produce identical output to the original.

Table 2 also presents the modifications made by NemesisGuard's instrumentation module to our application dataset. The column *Dir. Inst.* in the table represents the number of direct call and jump instructions that are statically rewritten by adjusting their target addresses. Also, column *Ind. Inst.* represents the number of indirect call and jump instructions redirected to the mapping routine for obtaining new targets dynamically. Additionally, data reports that NemesisGuard has an acceptable impact on binary size when delivering balancing instrumentation. Column *Size Inc.* represents the incurred size expansion on the code sections of instrumented application binaries. This overhead is positively correlated with the number of indirect branch instructions because NemesisGuard instruments them with mapping instructions. We do not include the mapping table in *.newtext* column in the table owing to the fact that it is always four times larger than the *.oldtext* column: mapping table keeps 4-byte data for every single byte in .oldtext.

6.4. Efficiency

In order to evaluate the impact of NemsisGuard instrumentation on the efficiency of IoT applications, we compare the execution time of an application with its instrumented version. Indeed, we start profiling all applications just before the main function begins execution and stops at a hard-coded point. Twenty runs of each application were averaged and in all runs the standard deviation was less than 2%. For example, we executed FatFS-uSD original and balanced application for formatting the SD card, creating a file, writing 2,048 bytes to the file, and verifying the contents of the file twenty times. Fig. 6 depicts the efficiency results. The average runtime overhead for the balanced application is 42%.

Furthermore, in Fig. 7, we illustrated how long it takes NemesisGuard to instrument and mitigate IoT applications against Nemesis side channel attacks. Indeed, large applications such as mbed-TLS take more time to be processed by NemesisGaurd. On average, NemesisGuard spends 304 s on applications in our dataset. We interpret this as a promising result which makes our framework a tool totally practical for mitigating Nemesis side channel attacks in the large-scale.

7. Related work

In this section, we discuss related work that are both complementary and orthogonal to our work including static binary instrumentation methods, software-based and microarchitectural side channel attacks, and the proposed methods for detecting and mitigating them.

Table 2
Statistics of IoT application binaries instrumented by NemesisGuard.

IoT Application	MCU	Dir. Inst.	Ind. Inst.	.oldtext (KB)	.newtext (KB)	Size Inc. (%)
Audio-Playback	STMF479I	5290	259	110	473	330
LCD_Display	STMF479I	2107	103	30	98	226
LCD_Animate	STMF479I	2097	103	30	97	223
FatFs_uSD	STMF407G	1654	79	21	48	128
TCP_Echo_Client	STMF479I	3728	132	52	172	230
TCP_Echo_Server	STMF479I	3566	132	51	173	239
UDP_Echo_Client	STMF479I	3471	132	49	172	251
UDP_Echo_Server	STMF479I	3381	130	48	176	266
Camera-USB	STMF479I	2906	161	41	137	134
mbed-TLS	STMF401RE	8256	341	116	416	258
PLC	STMF479I	1451	190	22	60	172

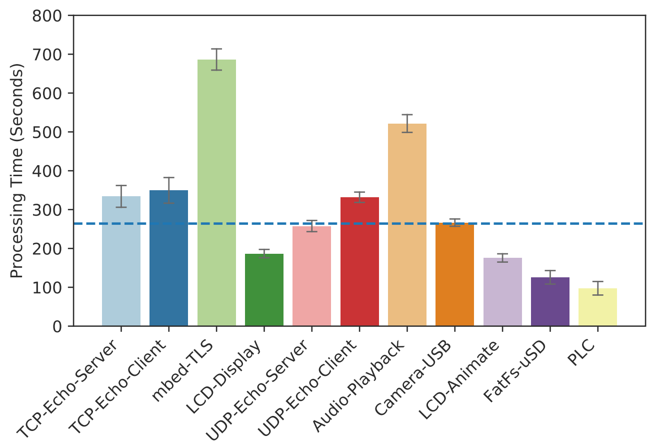


Fig. 7. NemesisGuard processing time for the IoT application binaries.

7.1. Static binary instrumentation

There are a number of static methods that transform binaries before execution. These methods differ from each other in how they transform binaries without breaking their functionality and semantics. For example, Uroboros [42] and Ramblr [41] present a set of heuristics to convert a binary into their own internal representations and perform instrumentation on those. However, heuristics-based approaches suffer from false positives and negatives that result in broken reassembled binary. RetroWrite [17] and Egalito [57] instrument executable binaries by leveraging relocation information which is only available in position independent codes. Unfortunately, this is not an applicable solution for IoT firmware that are mostly statically linked. Multiverse [58] proposes a new disassembling technique by disassembling instructions from every offset of code section to create a superset of all possible disassemblies. Multiverse binary rewriter is built upon this disassembler to instrument all superset instructions. As pointed out by Miller et el. [59], superset disassembly technique incurs a substantial code size overhead (763% on SPECint 2006 benchmarks). In addition, experimental results [57] show that Multiverse does not support statically linked binaries.

RevARM [40] is the only static binary rewriter that provides support for ARM architecture. However, RevARM is proposed for instrumenting mobile applications, not IoT applications which have significant resource constraints. Furthermore, RevARM uses unsound heuristic-based approach for rewriting binaries statically. Indeed, RevARM leverages a similar approach to Uroboros [42] for recognizing pointer-like data and identifying code pointers, an approach which is proved to be unsound in [41]. Moreover, RevARM instruments applications at a higher-level intermediate representation (IR). Lifting disassembly to a higher-level intermediate language requires precise modeling of instruction set architecture (ISA), which is an error-prone process.

7.2. Software-based side channel attacks

Ge et al. [23] conducted a comprehensive survey of attacks based on microarchitectural timing channels. However, all of these attacks are designed and developed for the modern processors with deployed advance mechanisms such as virtual memory addressing, caches and branch predictors. Thus, they are not practical for the embedded devices with limited resources and features [60]. Van Bulck et al. [13] were the first to propose remotely exploitable controlled-channel (i.e. Nemesis) for low-end embedded TEEs.

Similar to our work, there are a number of techniques that transform application codes for mitigating side channel attacks. For instance, SC-Eliminator [9] is a compiler-based method which takes a list of secrets in the code, and utilizes tag propagation to transform LLVM IR into its constant-time equivalent. Particularly, it mitigates cache timing channels by enforcing constant execution time for all secret-dependent operations. Molnar et al. [10] proposed a program counter security model in order to mitigate control-flow-based timing attacks. Precisely, this security model removes secret-dependent control flows by transforming application code through a modified compiler. Along similar lines, there are other approaches [11,12,16] proposed to mitigate timing side channel attacks by modifying a compiler to eliminate control-flow dependencies on secrets.

Unfortunately, above compiler-based approaches are impractical for COTS binaries and have not yet been widely adopted. This is mainly due to the fact that these approaches lift assembly code to a higher-level IR for rewriting; this process is known to be error-prone since it requires precise modeling of the Instruction Set Architecture (ISA) [17,18]. In other words, the process of lifting from disassembly to an IR is based on instruction semantics and needs to be implemented on a per-architecture basis. However, NemesisGuard does not need to lift disassembly code to a higher-level IR and works directly on disassembly generated from any off-the-shelf disassembler. Thus, in contrast to the previous approaches, NemesisGuard is a sound and complete approach for mitigating timing and interrupt latency side channel attacks. Additionally, NemesisGuard can be extended to support multiple architecture with small engineering efforts, which is not the case for previous proposed works.

Busi et al. [14,15] proposed a novel interrupt handling mechanism for Sancus TEE based on MSP430 architecture with the aim of mitigating Nemesis attacks. However, this method requires modifications to the hardware which is impractical for the devices that are actually deployed on the field. Pouyanrad et al. [44] designed and developed an automated tool to detect Nemesis vulnerabilities in MSP430 binaries. However, this mechanism is orthogonal to our work since it is only proposed for detecting Nemesis vulnerabilities and is not appropriate for mitigating them (see Table 3).

8. Discussion

Although NemesisGuard mitigates Nemesis side channel attacks effectively, there are still some challenges for future improvements. In

Table 3
A subjective comparison between NemesisGuard and state-of-the-art methods proposed for detecting and mitigating Nemesis side channel attacks- M: MSP430, A: ARM.

Objectives	Busi et al. [14]	Pouyanrad et al. [44]	Winderix et al. [16]	NemesisGuard
Architecture	M	M	M	A
Mitigation	✓	✗	✓	✓
Detection	✗	✓	✗	✗
Binary Support	✓	✓	✗	✓
w/o Hardware Modification	✗	✓	✓	✓

this section, we discuss the existing limitations in the current design and explore how they could be handled in the future.

The current design of NemesisGuard framework relies on static binary instrumentation to balance secret-dependent branches and mitigate Nemesis side channel attacks. Consequently, like other static binary instrumentation methods, NemesisGuard does not support dynamically loaded binaries. Indeed, support for instrumenting such binaries requires dynamic instrumentation methods which are orthogonal to this paper. In addition, NemesisGuard implementation is presently compatible with ARM architecture as the most widely used architecture for embedded systems [35]. However, since NemesisGuard is a platform-independent framework, it can support IoT applications developed for other architectures (e.g. x86) with a small extra engineering effort. As an instance, in order to support x86 architecture for instrumentation and balancing, it is mainly required to change the assembly language of the rewritten and inserted instructions and the mapping function.

The process of generating a precise CFG [38,61–64] can improve the performance of balancing procedure since the current design of NemesisGuard generates an over-approximated CFG to balance all secret dependent branches. Thus, having a more precise CFG removes the branches that are not actually taken at the runtime, making the balancing procedure faster. Nevertheless, NemesisGuard does not follow a precise recovery of the CFG as it is still an open problem and current solutions for this recovery may have false negatives (i.e. branches do not belong to the target sets according to the analysis, but are actually taken at the runtime). For effectively mitigating all Nemesis vulnerabilities in a binary, NemesisGuard can tolerate false positives but not false negatives. However, advancing the CFG generating procedure is out of scope of this paper, and we will leave them for future work.

Furthermore, NemesisGuard only mitigates interrupt latency side channel attacks (i.e. Nemesis) targeting IoT devices. Indeed, other instrumentation algorithms can be developed on top of NemesisGuard static binary instrumentation module for mitigating more types of software-based side channel attacks. However, mitigating other side channel attacks are out of scope of this paper, and we will leave them for future work.

9. Conclusion

Interrupt latency side channel attacks can lead to significant damage on IoT devices that are increasingly intertwined with critical industrial and medical processes. In this paper, we have proposed the first static binary instrumentation solution (called NemesisGuard) to automatically mitigate such attacks in IoT devices. NemesisGuard balances the execution time of corresponding instructions in secret-dependent branches by instrumenting them with compensation codes. In contrast to the state-of-the-art, NemesisGuard is a practical approach for COTS IoT binaries deployed in the field as it does not require a modified compiler or hardware. NemesisGuard framework is available as open-source at https://github.com/pwnforce/NemesisGuard.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, Belgium, and by the Flemish Research Programme Cybersecurity. We are grateful to anonymous reviewers for assisting us with their helpful comments and criticisms.

References

[1] Ericsson, Internet of things forecast, 2019, https://www.ericsson.com/en/mobility-report/internet-of-things-forecast. (Accessed February 2021).
[2] T. Alves, D. Felton, TrustZone: Integrated hardware and software security, ARM White Pap. 3 (4) (2004) 18–24.
[3] P. Koeberl, S. Schulz, A. Sadeghi, V. Varadharajan, TrustLite: A security architecture for tiny embedded devices, in: European Conference on Computer Systems, 2014, pp. 1–14.
[4] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens, Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base, in: USENIX Security Symposium, 2013.
[5] J. Noorman, J. Van Bulck, J. Tobias Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, F. Freiling, Sancus 2.0: A low-cost security architecture for IoT devices, ACM Trans. Priv. Secur. 20 (3) (2017) 1–33.
[6] P. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems, in: Annual International Cryptology Conference, Springer, 1996, pp. 104–113.
[7] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: Annual International Cryptology Conference, Springer, 1999, pp. 388–397.
[8] S. Jana, V. Shmatikov, Memento: Learning secrets from process footprints, in: IEEE Symposium on Security and Privacy, SP, IEEE, 2012, pp. 143–157.
[9] M. Wu, S. Guo, P. Schaumont, C. Wang, Eliminating timing side-channel leaks using program repair, in: ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2018, pp. 15–26.
[10] D. Molnar, M. Piotrowski, D. Schultz, 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.
[11] B. Coppens, I. Verbauwhede, K. De Bosschere, B. De Sutter, Practical mitigations for timing-based side-channel attacks on modern x86 processors, in: IEEE Symposium on Security and Privacy, SP, IEEE, 2009, pp. 45–60.
[12] J. Agat, Transforming out timing leaks, in: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2000, pp. 40–53.
[13] J. Van Bulck, F. Piessens, R. Strackx, Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic, in: ACM SIGSAC Conference on Computer and Communications Security, CCS, ACM, 2018, pp. 178–195.
[14] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. Mühlberg, F. Piessens, Provably secure isolation for interruptible enclaved execution on small microprocessors, in: Computer Security Foundations Symposium, CSF, IEEE, 2020, pp. 262–276.
[15] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. Mühlberg, F. Piessens, Provably secure isolation for interruptible enclaved execution on small microprocessors: Extended version, 2020, Preprint available at arXiv:2001.10881.
[16] H. Winderix, J. Mühlberg, F. Piessens, Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks.
[17] S. Dinesh, N. Burow, D. Xu, M. Payer, Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization, in: IEEE Symposium on Security and Privacy, SP, IEEE, 2020, pp. 1497–1511.
[18] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, M. Franz, BinRec: dynamic binary lifting and recompilation, in: European Conference on Computer Systems, EuroSys, 2020, pp. 1–16.
[19] Intel, Intel software guard extensions programming reference, 2013, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (Accessed May 2021).
[20] Samsung, Samsung KNOX, 2013, https://www.samsungknox.com/en. (Accessed May 2021).
[21] F. McKeen, I. Alexandrovich, A. Berenzon, C.V Rozas, H. Shafi, V. Shanbhogue, U.R. Savagaonkar, Innovative instructions and software model for isolated execution, HASP 10 (1) (2013).

[22] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, Foreshadow: Extracting the keys to the intel {sGX} kingdom with transient out-of-order execution, in: USENIX Security Symposium, 2018, pp. 991–1008.
[23] Q. Ge, Y. Yarom, D. Cock, G. Heiser, A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, J. Cryptogr. Eng. 8 (1) (2018) 1–27.
[24] D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of AES, in: Cryptographers' Track at the RSA Conference, Springer, 2006, pp. 1–20.
[25] C. Percival, Cache missing for fun and profit, in: BSDCan, 2005.
[26] F. Liu, Y. Yarom, Q. Ge, G. Heiser, R.B. Lee, Last-level cache side-channel attacks are practical, in: IEEE Symposium on Security and Privacy, SP, IEEE Computer Society, 2015, pp. 605–622.
[27] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, K. Römer, Hello from the other side: SSH over robust cache covert channels in the cloud., in: Network and Distributed System Security Symposium, Vol. 17, NDSS, 2017, pp. 8–11.
[28] T. Goodspeed, Practical attacks against the MSP430 BSL, in: Chaos Communications Congress, 2008.
[29] J.V. Cleemput, B. Coppens, B. De Sutter, Compiler mitigations for time attacks on modern x86 processors, ACM Trans. Archit. Code Optim. (TACO) 8 (4) (2012) 1–20.
[30] P. Puschner, R. Kirner, B. Huber, D. Prokesch, Compiling for time predictability, in: International Conference on Computer Safety, Reliability, and Security, Springer, 2012, pp. 382–391.
[31] A. Baumann, M. Peinado, G. Hunt, Shielding applications from an untrusted cloud with haven, ACM Trans. Comput. Syst. (TOCS) 33 (3) (2015) 1–26.
[32] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, VC3: Trustworthy data analytics in the cloud using SGX, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 38–54.
[33] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, J. Del Cuvillo, Using innovative instructions to create trustworthy software solutions, in: HASP@ ISCA, Vol. 11, 2013, pp. 2487726–2488370.
[34] G. Noubir, A. Sanatinia, Trusted code execution on untrusted platforms using intel SGX, Virus Bull. (2016).
[35] C.H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, D. Xu, Securing real-time microcontroller systems through customized memory view switching, in: Network and Distributed System Security Symposium, NDSS, 2018.
[36] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, H. Bos, An in-depth analysis of disassembly on full-scale x86/x64 binaries, in: USENIX Security Symposium, 2016, pp. 583–600.
[37] T. Bao, J. Burket, M. Woo, R. Turner, D. Brumley, { BYTEWEIGHT }: LEarning to recognize functions in binary code, in: USENIX Security Symposium, 2014, pp. 845–860.
[38] J. Kinder, H. Veith, Jakstab: A static analysis platform for binaries, in: International Conference on Computer Aided Verification, Springer, 2008, pp. 423–427.
[39] B. Schwarz, S. Debray, G. Andrews, Disassembly of executable code revisited, in: Working Conference on Reverse Engineering, IEEE, 2002, pp. 45–54.
[40] T. Kim, C.H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, D. Xu, RevARM: A platform-agnostic ARM binary rewriter for security applications, in: Annual Computer Security Applications Conference, ACSAC, 2017, pp. 412–424.
[41] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, G. Vigna, Ramblr: Making reassembly great again., in: Network and Distributed System Security Symposium, NDSS, 2017.
[42] S. Wang, P. Wang, D. Wu, Reassembleable disassembling, in: USENIX Security Symposium, 2015, pp. 627–642.
[43] S. Chen, Z. Lin, Y. Zhang, SelectiveTaint: Efficient data flow tracking with static binary rewriting, in: USENIX Security Symposium, 2021.
[44] S. Pouyanrad, J. Mühlberg, W. Joosen, SCFMSP: static detection of side channels in MSP430 programs, in: International Conference on Availability, Reliability and Security, ARES, 2020, pp. 1–10.
[45] ARM, ARMV7-M architecture reference manual, 2021, https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf. (Accessed May 2021).
[46] Capstone, Capstone: The ultimate disassembler framework, 2020, http://www.capstone-engine.org/, (Accessed May 2021).
[47] Eli Bendersky, Pyelftools: Parsing ELF and DWARF in python, 2012, https://github.com/eliben/pyelftools/. (Accessed May 2021).
[48] Quarkslab, Quarkslab Lief project, 2020, https://lief.quarkslab.com/. (Accessed May 2021).
[49] Pwntools, CTF framework and exploit development library, 2020, https://github.com/Gallopsled/pwntools/. (Accessed May 2021).
[50] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, G. Vigna, Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware., in: Network and Distributed System Security Symposium, Vol. 1, NDSS, 2015, p. 1.
[51] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, Sok:(state of) the art of war: Offensive techniques in binary analysis, in: IEEE Symposium on Security and Privacy, SP, IEEE, 2016.
[52] STMicroelectronics, STM32Cube MCU packages, 2021, https://www.st.com/en/embedded-software/stm32cube-mcu-mpu-packages.html. (Accessed November 2021).
[53] STMicroelectronics, STM32F401RE MCU, 2021, https://www.st.com/en/evaluation-tools/nucleo-f401re.html. (Accessed May 2021).
[54] STMicroelectronics, STM32f479I MCU, 2021, https://www.st.com/en/microcontrollers-microprocessors/stm32f469-479.html. (Accessed May 2021).
[55] STMicroelectronics, STM32F4DISCOVERY MCU, 2021, https://www.st.com/en/evaluation-tools/stm32f4discovery.html. (Accessed May 2021).
[56] STMicroelectronics, STM32L152RE MCU, 2021, https://www.st.com/en/evaluation-tools/nucleo-l152re.html. (Accessed May 2021).
[57] D. Williams-King, H. Kobayashi, K. Williams-King, G. Elaine Patterson, F. Spano, Y. Jian Wu, J. Yang, V. Kemerlis, Egalito: Layout-agnostic binary recompilation, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2020.
[58] E. Bauman, Z. Lin, K. Hamlen, Superset disassembly: Statically rewriting x86 binaries without heuristics, in: Proceedings of the Network and Distributed System Security Symposium, NDSS, 2018.
[59] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, Z. Lin, Probabilistic disassembly, in: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 1187–1198.
[60] M. Salehi, D. Hughes, B. Crispo, Microguard: Securing bare-metal microcontrollers against code-reuse attacks, in: IEEE Conference on Dependable and Secure Computing, DSC, IEEE, 2019, pp. 1–8.
[61] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, Z. Su, X-force: Force-executing binary programs for security applications, in: 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, pp. 829–844.
[62] C. Kruegel, W. Robertson, F. Valeur, G. Vigna, Static disassembly of obfuscated binaries, in: USENIX Security Symposium, Vol. 13, 2004, pp. 18–18.
[63] L. Harris, B. Miller, Practical analysis of stripped binary code, ACM SIGARCH Comput. Archit. News 33 (5) (2005) 63–68.
[64] C. Cifuentes, M. Van Emmerik, Recovery of jump table case statements from binary code, Sci. Comput. Programm. 40 (2–3) (2001) 171–188.

Majid Salehi is a Ph.D. researcher with the KU Leuven Computer Science Department, where he is a member of the imec-DistriNet research group. He received the M.Sc. degree from Sharif University of Technology, in 2016. His research interests include Internet of Things (IoT) security. He is particularly interested in issues concerning memory-based attacks in bare-metal embedded devices.

Gilles De Borger is an M.Sc. student in Computer Science at KU Leuven, Belgium, where he is working on IoT systems security. Other research interests include vulnerability detection and software testing.

Danny Hughes is a Professor with the Department of Computer Science of KU Leuven, Belgium, where he is a member of the imecDistriNet (Distributed Systems and Computer Networks) research group and leads the Networked Embedded Software taskforce. His current research is on distributed software systems and the Internet of Things.

Bruno Crispo holds a Ph.D. from Cambridge University, UK. He is full professor of computer science with the University of Trento, Italy, and visiting professor with KU Leuven, Belgium. His research interests include IoT security, network security, web security, biometric authentication and access control. He is an associate editor of the ACM Transactions on Privacy and Security and a senior member of the IEEE.