GAROTA: Generalized Active Root-Of-Trust Architecture (for Tiny Embedded Devices)

Esmerald Aliaj[1], Ivan De Oliveira Nunes[2], and Gene Tsudik[1]

[1]University of California, Irvine
[2]Rochester Institute of Technology
{ealiaj, gene.tsudik}@uci.edu, ivanoliv@mail.rit.edu

Abstract

Embedded (aka smart or IoT) devices are increasingly popular and becoming ubiquitous. Unsurprisingly, they are also attractive attack targets for exploits and malware. Low-end embedded devices, designed with strict cost, size, and energy limitations, are especially challenging to secure, given their lack of resources to implement sophisticated security services, available on higher-end computing devices. To this end, several tiny Roots-of-Trust (RoTs) were proposed to enable services, such as remote verification of device's software state and run-time integrity. Such RoTs operate **reactively**: they can prove whether a desired action (e.g., software update or program execution) **was** performed on a specific device. However, they can not guarantee that a desired action **will be** performed, since malware controlling the device can trivially block access to the RoT by ignoring/discarding received commands and other trigger events. This is an important problem because it allows malware to effectively "brick" or incapacitate a potentially huge number of (possibly mission-critical) devices.

Though recent work made progress in terms of incorporating more active behavior atop existing RoTs, much of it relies on extensive hardware support in the form of Trusted Execution Environments (TEEs), which are generally too costly for low-end devices. In this paper, we set out to systematically design a minimal active RoT for low-end MCU-s. We begin with three questions: *(1) What functionality is required to guarantee actions in the presence of malware? (2) How to implement this efficiently? and (3) What are the security benefits of such an active RoT architecture?* We then design, implement, formally verify, and evaluate *GAROTA* : Generalized Active Root-Of-Trust Architecture. We believe that *GAROTA* is the first clean-slate design of an active RoT for low-end MCU-s. We show how *GAROTA* guarantees that even a fully software-compromised low-end MCU performs a desired action. We demonstrate its practicality by implementing *GAROTA* in the context of three types of applications where actions are triggered by: sensing hardware, network events and timers. We also formally specify and verify *GAROTA* functionality and properties.

1 Introduction

The importance of embedded systems is hard to overestimate and their use in critical settings is projected to rise sharply [1]. Such systems are increasingly inter-dependent and used in many settings, including household, office, factory, automotive, health and safety, as well as national defense and space exploration. Embedded devices are usually deployed in large quantities and for specific purposes. Due to cost, size and energy constraints, they typically cannot host complex security mechanisms. Thus, they are an easy and natural target for attackers that want to quickly and efficiently cause harm on an organizational, regional, national or even global, level. Fundamental trade-offs between security and other priorities, such as cost or performance are a recurring theme in the domain of embedded devices. Resolving these trade-offs, is challenging and very important.

Numerous architectures focused on securing low-end micro-controller units (MCU-s) by designing small and affordable trust anchors [3]. However, most such techniques **operate passively**. They can prove, to a trusted party, that certain property (or action) is satisfied (or was performed) by a remote and potentially compromised low-end MCU. Examples of such services include remote attestation [6,10,15,23,34,42], proofs of remote software execution [17], control-flow & data-flow attestation [4, 19, 20, 43, 51, 55], as well as proofs of remote software update, memory erasure, and system-wide reset [5,9,16]. These architectures are typically designed to provide proofs that are unforgeable, despite potential compromise of the MCU.

Aforementioned approaches are passive in nature. While they can detect integrity violations of remote devices, they cannot guarantee that a given security or safety-critical task will be performed. For example, consider a network comprised of a large number (of several types of) simple IoT devices, e.g., an industrial control system. Upon detecting a large-scale compromise, a trusted remote controller wants to fix the situation by requiring all compromised devices to reset or erase themselves in order to expunge malware. Even if each device has an uncompromised, yet passive, RoT, malware (which is in full control of the device's software state) can easily intercept, ignore, or discard any requests for the RoT, thus preventing its functionality from being triggered. Therefore, the only way to repair these compromised devices requires direct physical access (i.e, reprogramming by a human) to each device. Beyond the DoS damage caused by the multitude of essentially "bricked" devices, physical reprogramming itself is slow and disruptive, i.e., a logistical nightmare.

Motivated by the above, some recent research [30, 53] yielded trust anchors with a more active behavior. Specifically, Xu et al. [53] propose the concept of Authenticated Watch-Dog Timers (WDT), which enforce periodic execution of a secure component (an RoT task), unless explicit authorization (which can itself include a set of tasks) is received from a trusted controller. In [30] this concept is realized with the reliance on ARM TrustZone, as opposed to a dedicated co-processor as in the original approach from [53]. Targeting lower-end devices, [38] considered the problem of guaranteeing periodic execution of a task. It refers to this important goal as "trusted scheduling", in the context of real-time operating systems. All these techniques [30, 38, 53] are time-based – they periodically and actively trigger RoT invocation, despite potential compromise of the host device. We discuss them in more detail in Section 5.4.

In this paper, we take the natural next step and design a more general active RoT, called *GAROTA*: Generalized Active Root-Of-Trust Architecture. Our goal is an architecture capable of triggering guaranteed execution of trusted and safety-critical tasks based on arbitrary events captured by hardware peripherals (including timers, GPIO ports, and network interfaces) of an MCU the software state of which may be currently compromised. In principle, any hardware event that causes an interrupt on the unmodified MCU can trigger guaranteed execution of trusted software in *GAROTA* (assuming proper configuration). In that vein, our work can be viewed as a generalization of concepts proposed in [30, 38, 53], enabling arbitrary events (interrupt signals, as opposed to the timer-based approach from prior work) to trigger guaranteed execution of trusted functionalities. In comparison, prior work has the advantage of relying on pre-existent hardware, thus not requiring any hardware changes. On the other hand, our clean-slate approach, based on a minimal hardware design, enables new applications and is applicable to lower-end resource-constrained MCU-s.

At a high level, *GAROTA* is based on two key concepts: "**Guaranteed Triggering**" and "**Re-Triggering on Failure**". The term trigger is used to refer to an event that causes *GAROTA* RoT to take over the execution in the MCU. The "guaranteed triggering" property ensures that a particular event of interest always triggers execution of *GAROTA* RoT. Whereas,"re-triggering on failure" assures that, if RoT execution is illegally interrupted for any reason (e.g., attempts to violate execution's integrity, power faults, or resets), the MCU resets and the RoT is guaranteed to be the first to execute after subsequent re-initialization. Figure 1 illustrates this workflow.

We use *GAROTA* to address three realistic and compelling use-cases for the active RoT:

- GPIO-TCB: A safety-critical sensor/actuator hybrid, which is guaranteed to sound an alarm if the sensed quantity (e.g., temperature, CO_2 level, etc) exceeds a certain threshold. This use-case exemplifies hardware-based triggering.
- TimerTCB: A real-time system where a predefined safety-critical task is guaranteed to execute periodically. This use-case exemplifies timer-based triggering, which is also attainable by [30, 38, 53].
- NetTCB: a trusted component that is always guaranteed to process commands received over the network, thus preventing malware in the MCU from intercepting and/or discarding commands destined for the RoT. This use-case exemplifies network-based triggering.

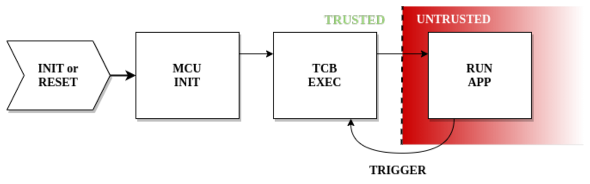


Figure 1: *GAROTA* Software Execution Flow

In all three cases, the guarantees hold even in case of full compromise of the MCU software state, as long as the RoT task itself is trusted.

In addition to designing and instantiating *GAROTA* with three use-cases, we formally specify *GAROTA* goals and requirements using Linear Temporal Logic (LTL). These formal specifications offer precise definitions for the security offered by *GAROTA* and its corresponding assumptions expected from the underlying MCU, i.e., its machine model. This can serve as an unambiguous reference for future implementations and for other derived services. Finally, we use formal verification to prove that the implementation of *GAROTA* hardware modules adheres to a set of sub-properties (also specified in LTL) that – when composed with the MCU machine model – are sufficient to achieve *GAROTA* end-to-end goals. In doing so, we follow a similar verification approach that has been successfully applied in the context of passive RoT-s [15, 17, 18].

We implement and evaluate *GAROTA* and make its verified implementation (atop the popular low-end MCU TI MSP430) along with respective computer proofs/formal verification publicly available in [2].

2 Scope

This work focuses on low-end embedded MCU-s and aims for a design with minimal hardware requirements. A minimal design simplifies reasoning about *GAROTA* and formally verifying its security properties. In terms of practicality and applicability, we believe that an architecture that is cost-effective enough for the lowest-end MCU-s can also be adapted (and potentially enriched) for implementations on higher-end devices with higher hardware budgets, while the other direction

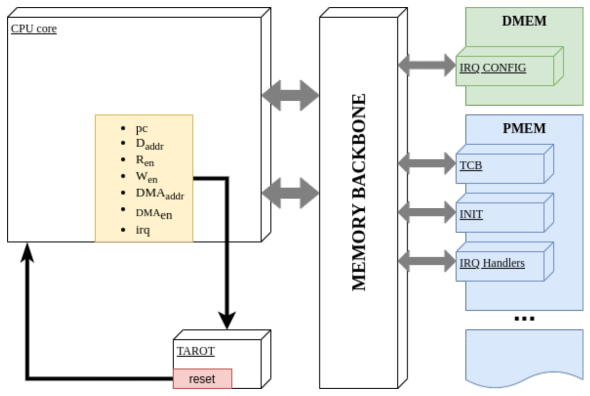


Figure 2: *GAROTA* in the MCU architecture

is usually more challenging. Thus, our design is applicable to the smallest and weakest devices based on low-power single-core platform with only a few KBytes of program and data memory (such as the aforementioned Atmel AVR ATmega and TI MSP430), with 8- and 16-bit CPUs, typically running at 1-16 MHz clock frequencies, with $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory ranging in size between 4 and 16 KBytes, while the rest of address space is available for program memory. Such devices usually run software atop "bare metal", execute instructions in place (physically from program memory), and have no memory management unit (MMU) to support virtual memory.

Our initial choice of implementing *GAROTA* atop the well-known TI MSP430 low-energy MCU is motivated by availability of an open-source version of the MSP430 hardware from OpenCores [28]. Nevertheless, *GAROTA* design and machine model are generic and should be applicable to other low-end MCU-s of the same class.

3 *GAROTA* Overview

The goal of *GAROTA* is to guarantee eventual execution of a pre-defined functionality $\mathcal{F}$ implemented as a trusted software executable. We refer to this executable as *GAROTA* trusted computing base (TCB). *GAROTA* is agnostic to the particular functionality implemented by $\mathcal{F}$, which allows guaranteed execution of arbitrary tasks, to be determined based on the application domain; see Section 5 for examples.

A trigger refers to a particular event that can be configured to cause the TCB to execute. Examples of possible triggers include hardware events from:

- *External (usually analog) inputs, e.g., detection of a button press, motion, sound or certain temperature/CO_2 threshold.*
- *Expiring timers, i.e., a periodic* trigger.
- *Arrival of a packet from the network, e.g., carrying a request to collect sensed data, perform sensing/actuation, or initiate a security task, such as erasing or resetting the device.*

If configured correctly, these events cause interrupts, which are used by *GAROTA* to guarantee execution of $\mathcal{F}$. Since trigger and TCB implementation are configurable, we assume that these initial configurations are done securely, at or before initial deployment. trigger configuration will include the types of interrupts and respective settings e.g., which GPIO port, what type of event, its time granularity, etc. At runtime, *GAROTA* protects the initial configuration from illegal modifications, i.e., ensures correct trigger behavior. This protection includes preserving interrupt configuration registers, interrupt handlers, and interrupt vectors. This way *GAROTA* guarantees that trigger always results in an invocation of the TCB.

However, guaranteed invocation of the TCB upon occurrence of a trigger is not sufficient to claim that $\mathcal{F}$ is properly performed, since the TCB code (and execution thereof) could itself be tampered with. To this end, *GAROTA* provides runtime protections that prevent any unprivileged/untrusted program from modifying the TCB code, i.e., the program memory region reserved for storing that code. (Recall that instructions execute in place, from program memory). *GAROTA* also monitors the execution of the TCB code to ensure:

1. **Atomicity:** Execution is atomic (i.e., uninterrupted), from the TCB's first instruction (legal entry), to its last instruction (legal exit);
2. **Non-malleability:** During execution, *DMEM* cannot be modified, other than by the TCB code itself, e.g., no modifications by other software or DMA controllers.

These two properties ensure that any potential malware residing on the MCU (i.e., compromised software outside TCB or compromised DMA controllers) cannot tamper with TCB execution.

GAROTA monitors execution and, whenever it detects a violation of *any* property (including atomicity, non-malleability, as well as any TCB code tampering or trigger misconfiguration) triggers an immediate MCU reset to a default trusted state wherein TCB code is the very first component to execute. Therefore, any attempt to interfere with the TCB functionality or execution causes the TCB to recover and re-execute, while guaranteeing that unprivileged/untrusted applications cannot interfere.

Both trigger configurations and the TCB implementation are updatable at run-time, as long as the updates are performed from within the TCB itself. While this feature is not strictly required for security, we believe it provides flexibility/updatability, while ensuring that untrusted software cannot modify *GAROTA* trusted components and configuration thereof. Section 4.7 discusses how *GAROTA* can enforce TCB confidentiality, which is applicable to cases where $\mathcal{F}$ implements cryptographic or privacy-sensitive tasks.

Each sub-property in *GAROTA* is implemented, and individually optimized, as a separate *GAROTA* sub-module. These sub-modules are then composed and shown secure (in the context of the MCU machine model) using a combination of model-checking-based formal verification and an LTL computer-checked proof. *GAROTA* modular design enables verifiability and minimality, resulting in low hardware overhead and significantly higher understanding and confidence about the security provided by its design and implementation.

As shown in Figure 2, *GAROTA* is implemented as a hardware component that monitors a set of CPU signals to detect violations to required security properties. As such it **does not** interfere with the CPU core implementation, e.g., by modifying its behavior or instruction set. In subsequent sections we describe these properties in more detail and discuss their implementation and verification. Finally, we use a commodity FPGA to synthesize *GAROTA* atop the low-end MCU MSP430 and report on its overhead.

4 *GAROTA* in Detail

We now get into the details of *GAROTA*. Section 4.1 provides some background on LTL and formal verification. Given some familiarity with these notions, it can be skipped without any loss of continuity.

4.1 LTL & Verification Approach

Formal Verification refers to the computer-aided process of proving that a system (e.g., hardware, software, or protocol) adheres to its well-specified goals. Thus, it assures that the system does not exhibit any unintended behavior, especially, in corner cases (rarely encountered conditions and/or execution paths) that humans tend to overlook.

To verify *GAROTA*, we use a combination of Model Checking and Theorem Proving, summarized next. In Model Checking, designs are specified in a formal computation model (e.g., as Finite State Machines or FSMs) and verified to adhere to formal logic specifications. The proof is performed through automated and exhaustive enumeration of all possible system states. If the desired specification is found not to hold for specific states (or transitions among them), a trace of the model that leads to the erroneous state is provided, and the implementation can then be fixed accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale well due to so-called "state explosion".

To cope with that problem, our verification approach (also used in prior work [15, 17]) is to specify each sub-property in *GAROTA* using Linear Temporal Logic (LTL) and verify each respective sub-module for compliance. In this process, the verification pipeline automatically converts digital hardware, described at Register Transfer Level (RTL) using Verilog, to Symbolic Model Verifier (SMV) [41] FSMs using Verilog2SMV [32]. The SMV representation is then fed to the well-known NuSMV [13] model-checker for verification against the specified LTL sub-properties. This automatic conversion ensures that no mistakes are introduced by otherwise manual translation from verification/proof models to actual hardware RTL. It thus represents another advantage of this verification pipeline over manual/paper proofs. Finally, the composition of the LTL sub-properties (verified in the model-checking phase) is proven to achieve *GAROTA* end-to-end goals using an LTL theorem prover [21]. This verification strategy is depicted in Figure 3.

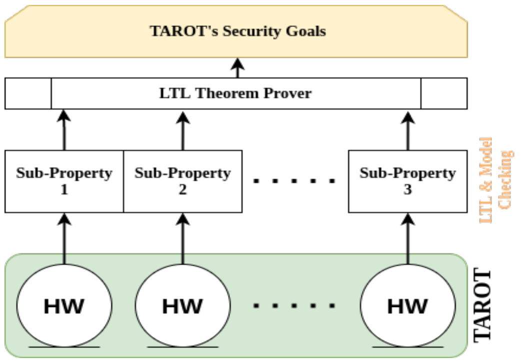


Figure 3: *GAROTA* verification strategy

Regular propositional logic includes propositional connectives, such as: conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and implication $\rightarrow$. LTL augments it with temporal quantifiers, thus enabling sequential reasoning. In this paper, we are interested in the following temporal quantifiers:

- $\mathbf{X}\phi$ – ne<u>X</u>t ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – <u>F</u>uture ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – <u>G</u>lobally ϕ: holds if for all future states ϕ is true.
- $\phi\ \mathbf{U}\ \psi$ – ϕ <u>U</u>ntil ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- $\phi\ \mathbf{W}\ \psi$ – ϕ <u>W</u>eak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. Or, more formally: $\phi\mathbf{W}\psi \equiv (\phi\mathbf{U}\psi) \vee \mathbf{G}(\phi)$

Note that, since *GAROTA* TCB is programmable and its code depends on the exact functionality $\mathcal{F}$ for each application domain, verification and correctness of any specific TCB code is not within our goals. We assume that the user is responsible for assuring correctness of the trusted code to be loaded atop *GAROTA* active RoT. This assumption is consistent with other programmable (though passive) RoTs, including those targeting higher-end devices, such as Intel SGX [31], and ARM TrustZone [8]. In many cases, we expect the TCB code to be minimal (see examples in Section 5), and thus less likely to have bugs.

As stated earlier, the verification strategy overviewed in this section was successfully used in prior work [15, 17] to verify *passive* RoTs (see Section 8). Nonetheless, that prior work did not consider availability properties, needed by an *active* RoT. Consequently, it did not require formal specification of properties that model guaranteed future actions upon specific `trigger`-s (see Definition 2). It also did not require cyclic specifications, such as re-triggering on failure (see Definition 3) to model the feature that a failed/tampered TCB execution is guaranteed to re-start.

PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access (read or write)
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
gie	Global Interrupt Enable: signal that indicates whether or not interrupts are globally enabled (1-bit).
irq	Signal that indicates if an interrupt is happening
$DMEM$	Region corresponding to the entire data memory of the MCU: $DMEM = [DMEM_{min}, DMEM_{max}]$.
$PMEM$	Region corresponding to the entire program memory of the MCU: $PMEM = [PMEM_{min}, PMEM_{max}]$.
TCB	Memory region reserved for the TCB's executable implementing $\mathcal{F}$. $TCB = [TCB_{min}, TCB_{max}]$. $TCB \in PMEM$.
$INIT$	Memory region containing the MCU's default initialization code. $INIT = [INIT_{min}, INIT_{max}]$. $INIT \in PMEM$.
$reset$	A 1-bit signal that reboots/resets the MCU when set to logical 1

Table 1: Notation Summary

In the same vein, prior work did not reason or axiomatize the same types of MCU behavior as *GAROTA*. In particular, most of the machine model in this paper (see Definition 1) is unique to *GAROTA*, except for LTL 1, that models CPU writes to memory, also modeled by prior work. Specifically, prior machine models did not axiomatize hardware behavior when: (1) a trigger/interrupt configuration is modified (LTL 2 from Definition 1); an interrupt is disabled by software (LTL 3 from Definition 1); or the effects of unmodifiable trigger initialization to the MCU runtime behavior (LTLs 4 and 5 from Definition 1).

Aforementioned definitions are unique requirements (in the machine model of Definition 1) and guarantees (in the goals of Definitions 2 and 3) of *GAROTA* that are proven to hold from composition of *GAROTA* sub-properties in Definition 4. Sub-properties are in turn offered by the formally verified implementation of *GAROTA* hardware. In the rest of this section, we detail the reasoning behind these axioms, property specifications, and their verifiable implementation and composition.

4.2 Notation, Machine Model, & Assumptions

This section discusses our machine and adversarial models. We start by overviewing them informally in Sections 4.2.1, 4.2.2 and 4.2.3). Then, Section 4.2.5, formalizes the machine model using LTL. For quick-reference, Table 1 summarizes the notation used in the rest of the paper.

4.2.1 CPU Hardware Signals

GAROTA neither modifies nor verifies the underlying CPU core/instruction set. It is assumed that the underlying CPU adheres to its specification and *GAROTA* is implemented as a standalone hardware module that runs in parallel with the CPU, and enforcing necessary guarantees in hardware. The following CPU signals are relevant to *GAROTA*:

H1 – *Program Counter (PC):* PC always contains the address of the instruction being executed in the current CPU cycle.

H2 – *Memory Address:* Whenever memory is read or written by the CPU, the data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

H3 – *DMA:* Whenever a DMA controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

H4 – *MCU Reset:* At the end of a successful reset routine, all registers (including PC) are set to zero before restarting normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. Once execution re-starts, PC is set to point to the first instruction in the boot section of program memory, referred to as $INIT$ (see M2 below). When a reset happens, the corresponding $reset$ signal is set. The same signal is also set when the MCU initializes for the first time. An MCU $reset$ also resets its DMA controller, and any prior configuration thereof. (DMA) behavior is configured by software at runtime. By default (i.e., after a reset) DMA is inactive.

H5 – *Interrupts:* Whenever an interrupt occurs, the corresponding irq signal is set. Interrupts may be globally enabled or disabled in software. The 1-bit signal gie always reflects whether or not they are currently enabled. The default gie state (i.e., at boot or after a reset) is disabled (logical zero).

4.2.2 Memory: Layout & Initial Configuration

As far as MCU initial memory layout and its initial software configuration (set at, or prior to, its deployment), the following are relevant to *GAROTA*:

M1 – *PMEM:* Corresponds to the entire $PMEM$ address space. Instructions are executed in place. Hence, at runtime, PC points to the $PMEM$ address storing the instruction being executed.

M2 – *INIT:* Section of $PMEM$ containing the MCU boot segment, i.e., the first software to be executed whenever the MCU boots or after a $reset$. We assume $INIT$ code is finite.

M3 – *TCB:* Section of $PMEM$ reserved for *GAROTA* trusted code, i.e., $\mathcal{F}$. TCB is located immediately after $INIT$; it is the first software to execute following successful completion of $INIT$.

M4 – *IRQ-Table and Handlers:* IRQ-Table is located in $PMEM$ and contains pointers to the addresses of so-called *interrupt handlers* (aka interrupt service routines or ISRs). When an interrupt occurs, the MCU hardware causes a jump to the corresponding handler routine. The address of this routine is specified by the IRQ-Table fixed index corresponding to that particular interrupt. Handler routines are code segments (functions) also stored in $PMEM$.

M5 – IRQ_{cfg}: Set of registers in $DMEM$ used to configure specific behavior of individual interrupts at runtime, e.g., deadline of a timer-based interrupt, or type of event on a hardware-based interrupt.

Note that any initial memory configuration could be changed at run-time (e.g., by malware that infects the device, as discussed in Section 4.2.4), unless it is explicitly protected by *GAROTA* verified hardware modules.

4.2.3 Initial Trigger Configuration

T1 – trigger: *GAROTA* trigger is configured, at MCU (pre)deployment-time, by setting the corresponding entry in IRQ-Table and respective handler to jump to the first instruction in TCB (TCB_{min}) and by configuring the registers in IRQ_{cfg} with desired interrupt parameters, reflecting the desired trigger behavior; see Section 5 for examples. Thus, a trigger event causes the TCB code to execute, as long as the initial configuration is maintained.

The initial trigger configuration is not much different from a regular interrupt configuration in a typical embedded system program. The trigger index in IRQ-Table must correctly point to *GAROTA* TCB legal entry point, just as regular interrupts must correctly point to their respective handler entry points. For example, to initially configure a timer-based trigger, the address in IRQ-Table corresponding to the respective hardware timer is set to point to TCB_{min} and the correspondent registers in IRQ_{cfg} are set to define the timer deadline and thus the desired interrupt period.

4.2.4 Adversarial Model

We consider an adversary $\mathcal{A}$dv that controls MCU's entire software state, including code, and data. $\mathcal{A}$dv can read-/write from/to any memory that is not explicitly protected by hardware-enforced access control rules. $\mathcal{A}$dv might also have full control over the Direct Memory Access (DMA) controller in the MCU. Recall that DMA allows a hardware controller to directly access main memory ($PMEM$ or $DMEM$) without going through the CPU.

Physical Attacks: physical and hardware-focused attacks are out of the scope of *GAROTA*. Specifically, we assume that $\mathcal{A}$dv can not modify hardware, induce hardware faults, or interfere with *GAROTA* via physical presence attacks and/or side-channels. Protection against such attacks is an orthogonal issue that we defer to future work. For an overview of potential countermeasures using physical security and tamper resistance techniques see [44, 46].

Network DoS Attacks: we also consider out-of-scope network DoS attacks, whereby $\mathcal{A}$dv drops traffic to/from MCU, or floods MCU with traffic, or simply jams communication. Note that this assumption is relevant only to network-triggered events, exemplified by the NetTCB instantiation of *GAROTA*, described in Section 5.3.

Note that network DoS attacks could be detected by orthogonal techniques, such as network traffic monitoring [37, 54, 56, 57]. Once detected, out-of-band measures can be taken to mitigate such attacks. *GAROTA* is particularly concerned with malware that infects the MCU and ignores commands/trigger events, even when commands arrive and triggers occur. Hence, network DoS monitoring is a complementary measure to ensure delivery of packets to the MCU (in *GAROTA* NetTCB use-case), while *GAROTA* can ensure that received commands are indeed processed.

Correctness of TCB's Executable: we stress that the purpose of *GAROTA* is guaranteed execution of $\mathcal{F}$, as specified by the application developer and loaded onto *GAROTA* TCB at deployment time. Similar to existing RoTs (e.g., TEE-s in higher-end CPUs) *GAROTA* does **not** check correctness of, and absence of implementation bugs in, $\mathcal{F}$'s implementation. In many applications, $\mathcal{F}$ code is minimal; see examples in Section 5. Moreover, correctness of $\mathcal{F}$ need **not** be assured locally. Since embedded applications are originally developed on more powerful devices (e.g., general-purpose computers), various vulnerability detection methods, e.g., fuzzing [12], static analysis [14], or formal verification, can be employed to avoid or detect implementation bugs in $\mathcal{F}$. All that can be performed off-line before loading $\mathcal{F}$ onto *GAROTA* TCB and the entire issue is orthogonal to *GAROTA* functionality.

4.2.5 Machine Model (Formally)

Based on the high-level properties discussed earlier in this section, we now formalize the subset (relevant to *GAROTA*) of the MCU machine model using LTL. Figure 4 presents our machine model as a set of LTL statements.

LTL statement (1) models the fact that modifications to a given memory address (X) can be done either via the CPU or DMA. Modifications by the CPU imply setting $W_{en} = 1$ and $D_{addr} = X$. If X is a memory region, rather than a single address, we denoted that a modification happened within the particular region by saying that $D_{addr} \in X$, instead. Conversely, DMA modifications to region X require $DMA_{en} = 1$ and $DMA_{addr} \in X$. This models the MCU behaviors stated informally in **H2** and **H3**.

In accordance with **M4** and **M5**, a successful modification to a pre-configured trigger implies changing interrupt tables, interrupt handlers, or interrupt configuration registers (ICR-s). Since, per **M4**, the first two are located in $PMEM$, modifying them means writing to $PMEM$. The ICR is located in a

Definition 1. Machine Model:

Memory Modifications:

$$\mathbf{G}:\{\mathsf{modMem}(X) \rightarrow (W_{en} \wedge D_{addr} \in X) \vee (DMA_{en} \wedge DMA_{addr} \in X)\} \tag{1}$$

Successful Trigger Modification:

$$\mathbf{G}:\{\mathsf{mod}(\mathsf{trigger}_{cfg}) \rightarrow [(\mathsf{modMem}(PMEM) \vee \mathsf{modMem}(IRQ_{cfg})) \wedge \neg reset]\} \tag{2}$$

Successful Interrupt Disablement:

$$\boldsymbol{G}\!:\!\{\mathsf{disable}(irq) \rightarrow [\neg reset \wedge gie \wedge \neg\mathbf{X}(gie) \wedge \neg\mathbf{X}(reset)]\} \tag{3}$$

Trigger/TCB Initialization (4 & 5):

$$\boldsymbol{G}\!:\!\{\neg\mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\} \wedge \boldsymbol{G}\!:\!\{\neg\mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\} \rightarrow \boldsymbol{G}\!:\!\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\} \tag{4}$$

$$\boldsymbol{G}\!:\!\{\neg\mathsf{modMem}(PMEM) \vee PC \in TCB\} \rightarrow \boldsymbol{G}\!:\!\{reset \rightarrow F(PC = TCB_{min})\} \tag{5}$$

Figure 4: MCU machine model (subset) in LTL.

DMEM location denoted IRQ_{cfg}. Therefore, the LTL statement (2) models a successful misconfiguration of trigger as requiring a memory modification either within *PMEM* or within IRQ_{cfg}, without causing an immediate system-wide reset ($\neg reset$). This is because an immediate reset prevents the modification attempt from taking effect (see **H4**).

LTL (3) models that attempts to disable interrupts are reflected by *gie* CPU signal (per **H5**). In order to successfully disable interrupts, $\mathcal{A}$dv must be able to switch interrupts from enabled ($gie = 1$) to disabled ($\neg\mathbf{X}(gie)$ – disabled in the following cycle), without causing an MCU *reset*.

Recall that (from **H1**) *PC* reflects the address of the instruction currently executing. $PC \in TCB$ implies that *GAROTA* TCB is currently executing. LTL (4) models **T1**. As long as the initial proper configuration of trigger is never modifiable by untrusted software ($\mathbf{G}$:$\{\neg\mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\}$) and that untrusted software can never globally disable interrupts ($\mathbf{G}$:$\{\neg\mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\}$), a trigger would always cause TCB execution ($\mathbf{G}$:$\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\}$). Recall that we assume that the TCB may update – though not misconfigure – trigger behavior, since the TCB is trusted. Similarly, LTL 5 states that, as long as *PMEM* is never modified by untrusted software, a *reset* will always trigger TCB execution (per **H4**, **M2**, and **M3**).

4.3 *GAROTA* End-To-End Goals Formally

Using the notation from Section 4.2, we proceed with the formal specification of *GAROTA* end-goals in LTL. Definition 2 specifies the "guaranteed trigger" property. It states in LTL that, whenever a trigger occurs, a TCB execution/invocation (starting at the legal entry point) will follow.

While Definition 2 guarantees that a particular interrupt of interest (trigger) will cause the TCB execution, it does not guarantee proper execution of the TCB code as a whole. The "re-trigger on failure" property (per Definition 3) stipulates that, whenever TCB starts execution (i.e., $PC \in TCB$), it must execute without interrupts or DMA interference [1], i.e., $\neg irq \wedge \neg dma_{en} \wedge PC \in TCB$. This condition must hold until:

1. $PC = TCB_{max}$: the legal exit of TCB is reached, i.e., execution concluded successfully.

   OR

2. $F(PC = TCB_{min})$: another TCB execution (from scratch) has been triggered to occur.

In other words, this specification reflects a cyclic requirement: either the security properties of the TCB proper execution are not violated, or TCB execution will re-start later.

Note that we use the quantifier Weak Until (**W**) instead regular Until (**U**), because, for some embedded applications, the TCB code may execute indefinitely; see Section 5.1 for an example.

4.4 *GAROTA* Sub-Properties

Based on our machine model and *GAROTA* end goals, we now postulate a set of necessary sub-properties to be implemented by *GAROTA*. Next, Section 4.5 shows that this minimal set of sub-properties suffices to achieve *GAROTA* end-to-end goals with a computer-checked proof. LTL specifications of the sub-properties are presented in Figure 6.

GAROTA enforces that only trusted updates to *PMEM* are allowed at runtime. *GAROTA* hardware issues a system-wide MCU *reset* upon detecting any attempt to modify *PMEM* at runtime, unless this modification comes from the execution of the TCB code itself. This property is formalized in LTL (6). It prevents any untrusted application software from misconfiguring IRQ-Table and interrupt handlers, as well as from modifying the *INIT* segment and the TCB code itself, because these sections are located within *PMEM*. As a side benefit, it also prevents attacks that attempt to physically wear

[1] Since DMA could tamper with intermediate state/results in *DMEM*.

Definition 2. Guaranteed Trigger:

$$\textbf{\textit{G:}}\{\text{trigger} \rightarrow \mathbf{F}(PC = TCB_{min})\}$$

Definition 3. Re-Trigger on Failure:

$$\textbf{\textit{G:}}\{PC \in TCB \rightarrow [\ (\neg irq \wedge \neg dma_{en} \wedge PC \in TCB) \quad \textbf{\textit{W}} \quad (PC = TCB_{max} \vee \textbf{\textit{F}}(PC = TCB_{min})\ ]\}$$

Figure 5: Formal Specification of *GAROTA* end-to-end goals.

Definition 4. *LTL Sub-Properties implemented & enforced by GAROTA.*
Trusted** PMEM **Updates:

$$\mathbf{G}: \ \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in PMEM)] \vee [DMA_{en} \wedge (DMA_{addr} \in PMEM)] \rightarrow reset\} \quad (6)$$

IRQ Configuration Protection:

$$\mathbf{G}: \ \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in IRQ_{cfg})] \vee [DMA_{en} \wedge (DMA_{addr} \in IRQ_{cfg})] \rightarrow reset\} \quad (7)$$

Interrupt Disablement Protection:

$$\mathbf{G}: \ \{\neg reset \wedge gie \wedge \neg \textbf{\textit{X}}(gie) \rightarrow (\textbf{\textit{X}}(PC) \in TCB) \vee \textbf{\textit{X}}(reset)\} \quad (8)$$

TCB Execution Protection:

$$\mathbf{G}: \ \{\neg reset \wedge (PC \in TCB) \wedge \neg(\mathbf{X}(PC) \in TCB) \rightarrow PC = TCB_{max} \vee \ \mathbf{X}(reset)\ \} \quad (9)$$

$$\mathbf{G}: \ \{\neg reset \wedge \neg(PC \in TCB) \wedge (\mathbf{X}(PC) \in TCB) \rightarrow \mathbf{X}(PC) = TCB_{min} \vee \ \mathbf{X}(reset)\} \quad (10)$$

$$\mathbf{G}: \ \{(PC \in TCB) \wedge (irq \vee dma_{en}) \rightarrow reset\} \quad (11)$$

Figure 6: Formal specification of sub-properties verifiably implemented by *GAROTA* hardware module.

off Flash (often used to implement *PMEM* in low-end devices) by excessively and repeatedly overwriting it at runtime [7]. Similarly, *GAROTA* prevents untrusted components from modifying $IRQ_{cfg} - DMEM$ registers controlling the trigger configuration. This is specified by LTL 7.

Remark: *local updates, via direct physical connection (e.g., J-TAG or USB) to the MCU are still possible and unaffected by GAROTA, because GAROTA hardware protection is only active at runtime. If remote updates at runtime are desirable, they must be supported as part of GAROTA TCB. See Section 7 for a discussion on supporting remote updates.*

LTL 8 enforces that interrupts can not be globally disabled by untrusted applications. Since, each trigger is based on interrupts, disablement of all interrupts would allow untrusted software to disable the trigger itself, and thus the active behavior of *GAROTA*. This requirement is specified by checking the relation between current and next values of *gie*, using the LTL ne**X**t operator. In order to switch *gie* from logical 0 (current cycle) to 1 (next cycle), TCB must be executing when *gie* becomes 0 ($\mathbf{X}(PC) \in TCB$)), or the MCU will *reset*.

To assure that TCB code is invoked and executed properly, *GAROTA* hardware implements LTL-s (9), (10), and (11). LTL 9 enforces that the only way for *TCB*'s execution to terminate, without causing a *reset*, is through its last instruction (its only legal exit): $PC = TCB_{max}$. This is specified by checking the relation between current and next *PC* values using LTL ne**X**t operator. If the current *PC* value is within *TCB*, and next *PC* value is outside *TCB*, then either current *PC* value must be the address of TCB_{max}, or *reset* is set to 1 in the next cycle. Similarly, LTL 10 enforces that the only way for *PC* to enter *TCB* is through the very first instruction: TCB_{min}. This prevents *TCB* execution from starting at some point in the middle of *TCB*, thus making sure that *TCB* always executes in its entirety. Finally, LTL 11 enforces that *reset* is always set if interrupts or DMA modifications happen during *TCB*'s execution. Even though LTLs 9 and 10 already enforce that PC can not change to anywhere outside *TCB*, interrupts could be programmed to return to an arbitrary instruction within the *TCB*. Or, DMA could change *DMEM* values currently in use by TCB. Both of these events can alter TCB behavior and are treated as violations.

Next, Section 4.5 presents a computer-checked proof of sufficiency of this set of sub-properties to imply *GAROTA* end-to-end goals. Then, Section 4.6 presents FSM-s from our Verilog implementation, that are formally verified to correctly implement each requirement.

4.5 *GAROTA* Composition Proof

GAROTA end-to-end sufficiency is stated in Theorems 1 and 2. The complete computer-checked proofs (using Spot2.0 [21]) of Theorems 1 and 2 are publicly available at [2]. Below we

Theorem 1. *Definition 1 $\wedge$ LTLs 6,7,8 $\rightarrow$ Definition 2.*

Theorem 2. *Definition 1 $\wedge$ LTLs 6,9,10,11 $\rightarrow$ Definition 3.*

present the intuition behind them.

Proof of Theorem 1 (Intuition). From machine model LTL (4), as long as the (1) initial trigger configuration is never modified from outside the TCB; and (2) interrupts are never disabled from outside the TCB; it follows that a trigger will cause a proper invocation of the TCB code. Also, successful modifications to the trigger's configuration imply writing to $PMEM$ or IRQ_{cfg} without causing a $reset$ (per LTL (2)). Since *GAROTA* verified implementation guarantees that memory modifications (specified in LTL (1)) to $PMEM$ (LTL (6)) or IRQ_{cfg} (LTL (7)) always cause a $reset$, illegal modifications to $trigger_{cfg}$ are never successful. Finally, LTL (8) assures that any illegal interrupt disablement always causes a $reset$, and is thus never successful). Therefore, *GAROTA* satisfies all necessary conditions to guarantee the goal in Definition 2. □

Proof of Theorem 2 (Intuition). The fact that a reset always causes a later call to the TCB follows from the machine model's LTL (5) and *GAROTA* guarantee in LTL (6). LTLs (9) and (9) ensure that the TCB executable is properly invoked and executes atomically, until its legal exit. Otherwise a $reset$ flag is set, which (from the above argument) implies a new call to TCB. Finally, LTL 11 assures that any interrupt or DMA activity during TCB execution will cause a $reset$, thus triggering a future TCB call and satisfying Definition 3. □

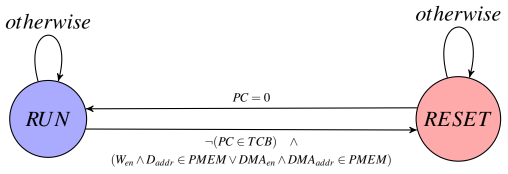
See [2] for the formal computer-checked proofs.

4.6 Sub-Module Implementation+Verification

Following the sufficiency proof in Section 4.5 for sub-properties in Definition 4, we proceed with the implementation and formal verification of *GAROTA* hardware using the NuSMV model-checker (see Section 4.1 for details).

GAROTA modules are implemented as Mealy FSMs (where outputs change with the current state and current inputs) in Verilog. Each FSM has one output: a local $reset$. *GAROTA* output $reset$ is given by the disjunction (logic *or*) of local $reset$-s of all sub-modules. Thus, a violation detected by any sub-module causes *GAROTA* to trigger an immediate MCU $reset$. For the sake of easy presentation we do not explicitly represent the value of $reset$ in the figures. Instead, we define the following implicit representation:

1. $reset$ output is 1 whenever an FSM transitions to the $RESET$ state (represented in red color);
2. $reset$ output remains 1 until a transition leaving the $RESET$ state is triggered;



otherwise
otherwise
RUN
RESET
$PC = 0$
$\neg(PC \in TCB) \quad \wedge$
$(W_{en} \wedge D_{addr} \in PMEM \vee DMA_{en} \wedge DMA_{addr} \in PMEM)$

Figure 7: Verified FSM for LTL 6.

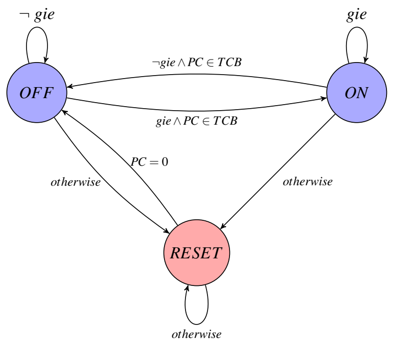


Figure 8: Verified FSM for LTL 8.

3. $reset$ output is 0 in all other states (represented in blue color).

Note that all FSM-s remain in the $RESET$ state until $PC = 0$, which signals that the MCU reset routine finished.

Figure 7 illustrates *GAROTA* sub-module responsible for assuring that $PMEM$ modifications are only allowed from within the TCB. This minimal 2-state machine works by monitoring PC, W_{en}, D_{addr}, DMA_{en}, and DMA_{addr} to detect illegal modification attempts by switching from RUN to $RESET$ state, upon detection of any such action. It is verified to adhere to LTL (6). A similar FSM is used to verifiably enforce LTL (7), with the only distinction of checking for writes within IRQ_{cfg} region instead, i.e., $D_{addr} \in IRQ_{cfg}$ and $DMA_{addr} \in IRQ_{cfg}$. Given the similarity, we omit the illustration of this FSM.

Figure 8 presents an FSM implementing LTL 8. It monitors the "global interrupt enable" (gie) signal to detect attempts to illegally disable interrupts. It consists of three states: (1) ON, representing execution periods where $gie = 1$; (2) OFF, for cases where $gie = 0$, and (3) $RESET$. To switch between ON and OFF states, this FSM requires $PC \in TCB$, thus preventing misconfiguration by untrusted software.

Finally, the FSM in Figure 9 verifiably implements LTLs 9, 10, and 11. This FSM has 5 states, one of which is $RESET$. Two basic states correspond to whenever: the TCB is executing (state "$\in TCB$"), and not executing (state "$\notin TCB$"). From $\notin TCB$ the only reachable path to $\in TCB$ is through state TCB_{entry}, which requires $PC = TCB_{min}$ – TCB only legal entry point. Similarly, from $\in TCB$ the only reachable path to $\notin TCB$ is through state TCB_{exit}, which requires $PC = TCB_{max}$

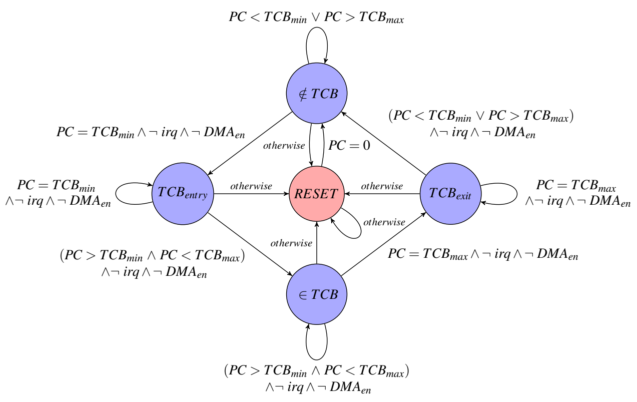

$PC < TCB_{min} \vee PC > TCB_{max}$
$\notin TCB$
$PC = TCB_{min} \wedge \neg irq \wedge \neg DMA_{en}$
$(PC < TCB_{min} \vee PC > TCB_{max})$ $\wedge \neg irq \wedge \neg DMA_{en}$
otherwise
$PC = 0$
$PC = TCB_{min}$ $\wedge \neg irq \wedge \neg DMA_{en}$
TCB_{entry}
otherwise
RESET
otherwise
TCB_{exit}
$PC = TCB_{max}$ $\wedge \neg irq \wedge \neg DMA_{en}$
otherwise
otherwise
$(PC > TCB_{min} \wedge PC < TCB_{max})$ $\wedge \neg irq \wedge \neg DMA_{en}$
$PC = TCB_{max} \wedge \neg irq \wedge \neg DMA_{en}$
$\in TCB$
$(PC > TCB_{min} \wedge PC < TCB_{max})$ $\wedge \neg irq \wedge \neg DMA_{en}$

Figure 9: Verified FSM for LTLs 9–11.

– TCB only legal exit. Also, in all states where $PC \in TCB$ (including entry and exit transitions) this FSM requires DMA and interrupts to remain inactive. Any violation of these requirements, in any of the four regular states, causes the FSM transition to $RESET$, thus enforcing TCB execution protection.

4.7 TCB Confidentiality

Confidentiality of TCB data and code with respect to untrusted applications is of particular interest when $\mathcal{F}$ implements cryptographic functions or privacy-sensitive tasks.

This goal can be achieved by including and epilogue phase in the TCB executable, with the goal of performing a $DMEM$ cleanup, erasing all traces of the TCB execution from the stack and heap. While the TCB execution may be interrupted before the execution of the epilogue phase, such an interrupt will cause an MCU $reset$. The *Re-Trigger on Failure* property assures that TCB code will execute (as a whole) after any $reset$ and will thus erase remaining execution traces from $DMEM$ before subsequent execution of untrusted applications. In a similar vein, if confidentiality of the executable is desirable, it can be implemented following LTL (12), which formalizes read attempts based on R_{en} signal:

$$\begin{aligned}
&\mathbf{G}: \{ \\
&\quad [\neg(PC \in TCB) \wedge R_{en} \wedge (D_{addr} \in TCB) \vee \\
&\quad DMA_{en} \wedge (DMA_{addr} \in TCB)] \rightarrow reset \\
&\}
\end{aligned} \tag{12}$$

An FSM implementing this property is shown in Figure 10. Note that, despite visual similarity with the FSM in Figure 7, the confidentiality FSM checks for *reads* (instead of writes) to the TCB (instead of entire $PMEM$).

This property prevents external reads to TCB code by monitoring R_{en}, D_{addr}, and DMA. When combined with the aforementioned erasure epilogue, it also enables secure storage

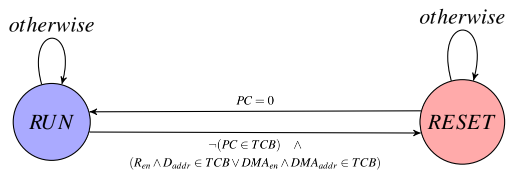

otherwise
otherwise
$PC = 0$
RUN
RESET
$\neg(PC \in TCB) \wedge$
$(R_{en} \wedge D_{addr} \in TCB \vee DMA_{en} \wedge DMA_{addr} \in TCB)$

Figure 10: Verified FSM for LTL 12.

of cryptographic secrets within the TCB binary, as in other architectures, e.g., [5,6,22]. This part of *GAROTA* design is optional, since some embedded applications do not require confidentiality, as discussed in Sections 5.1 and 5.2.

4.8 Resets & Availability

One important remaining issue is *availability*. For example, malware might interrupt (or tamper with) with $INIT$ execution after a $reset$ preventing the subsequent execution of TCB. Also, malware could to interrupt the TCB execution, after each *re*-trigger, with the goal of resetting the MCU indefinitely, and thereby preventing TCB execution from ever completing its task.

We observe that (under *GAROTA* assumptions) such actions are not possible, since they would require either DMA activity or interrupts to: (1) hijack $INIT$ control-flow; or (2) abuse *GAROTA* to successively $reset$ the MCU during TCB execution after each re-trigger. Given **H5** interrupts are disabled by default at boot time. Additionally, **H4** states that any prior DMA configuration is cleared to the default disabled state after a $reset$. Hence, $INIT$ and the first execution of TCB after a $reset$ cannot be interrupted or tampered with by DMA.

Finally, we note that, despite preventing security violations by (and implementing re-trigger based on) resetting the MCU, *GAROTA* does not provide any advantage to malware that aims to simply disrupt execution of (non-TCB) applications by causing $resets$. Any software running on bare metal (including malware) can always intentionally reset the MCU. Resets are the default mechanism to recover from regular software faults on unmodified (off-the-shelf) low-end MCU-s, regardless of *GAROTA*.

5 Sample Applications

Many low-end MCU use-cases and applications can benefit from trigger-based active RoTs. To demonstrate generality of *GAROTA*, we prototyped three concrete examples, each with a different type of trigger. This section overviews these examples: (1) GPIO-TCB uses external analog events (Section 5.1), TimerTCB uses timers (Section 5.2), and NetTCB uses network events (Section 5.3). Finally, Section 5.4 discusses how *GAROTA* can match active RoT services proposed in [53], [30], and [38].

```
1  int main() {
2     TCB();
3     main_loop();
4     return 0;
5  }
```

Figure 11: Program Entry Point

```
1  void setup (void) {
2     P1DIR  = 0x00;
3     P1IE   = 0x01;
4     P1IES  = 0x00;
5     P1IFG  = 0x00;
6  }
```

Figure 12: Trigger Setup

5.1 GPIO-TCB: Critical Sensing+Actuation

Our first application example, GPIO-TCB, operates in the context of a safety-critical temperature sensor. In this setting, we want to use *GAROTA* to assure that the sensor's most safety-critical function – *sounding an alarm* – is never prevented from executing due to software compromise of the underlying MCU. Once the alarm is on, the TCB implementation will also send a help message through the MCU UART communication interface[2] and keep the alarm on until someone physically presses a particular button to shut the alarm down. This example also illustrates the use-case where the TCB code awaits for an asynchronous input before resuming execution of untrusted software. We use a standard built-in MCU interrupt, based on General Purpose Input/Output (GPIO) to implement trigger. Since this is our first example, we discuss GPIO-TCB in more detail than the other two.

As shown in Figure 11, MCU execution always starts by calling the TCB (at line 2). Therefore, after MCU initialization/reset, unprivileged (non-TCB) applications can only execute after the TCB; assuming, of course, that formal guarantees discussed in Section 4 hold. These (unprivileged) applications are implemented inside *main_loop* function (at line 3).

The correct trigger configuration in GPIO-TCB can be achieved in two ways. The first way is to set IRQ_{cfg} to the desired parameters at MCU deployment time, by physically writing this configuration to IRQ_{cfg}. The second option is to implement this configuration in software as a part of the TCB. Since the TCB is always the first to run after initialization/reset, it will configure IRQ_{cfg} correctly, enabling subsequent trigger-s at runtime.

Figure 12 exemplifies IRQ_{cfg} configuration, implemented as part of the TCB, i.e., called from within the TCB. This `setup` function is statically linked to be located inside the TCB memory region, thus respecting "TCB Execution Protection" LTL rules (see Definition 4). The setup first configures

[2]UART is the main communication interface in the MCU. It can in turn connect to a ZigBee or Bluetooth radio that relays the message wirelessly, as described in [25].

```
1  ISR(PORT1, TCB) {
2     dint();
3     if (first_run) {
4        setup();
5     }
6
7     P3DIR = 0x01; // set P3 as output
8     P3OUT = 0x01; // turn buzzer on
9
10    UART_BAUD = BAUD; // set UART to default speed (BAUD macro)
11    UART_CTL  = UART_EN; // turn UART on;
12
13     while (UART_STAT & UART_TX_FULL); //wait for any previous send to complete
14     UART_TXD = 'H';                   //send 1st byte
15     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
16     UART_TXD = 'E';                   //send 2nd byte
17     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
18     UART_TXD = 'L';                   //send 3rd byte
19     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
20     UART_TXD = 'P';                   //send 4th byte
21
22    P2DIR = 0x00; // set P2 GPIO as input
23    while (1) { // wait for button press
24        if (P2IN == 0x01) { // button press detected
25            break;
26        }
27    }
28    P3OUT = 0x00; // turn buzzer off
29    eint();
30    return();
31 }
```

Figure 13: GPIO-TCB use-case example

the physical port $P1$ as an input (line 2, "P1 direction" set to 0x00, whereas 0x01 would set it as an output). At line 3, $P1$ is set as "interrupt-enabled" ($P1IE = 0x01$). A value of $P1IES = 0x00$ (line 4) indicates that, if the physical voltage input of $P1$ changes from logic 0 to 1 ("low-to-high" transition), a GPIO interrupt is triggered and the respective handler is called. Finally, $P1IFG$ is cleared to indicate that the MCU is free to receive interrupts (as opposed to busy). This initial trusted configuration of IRQ_{cfg} cannot be modified afterwards by untrusted applications due to *GAROTA* guarantees (see Section 4). Based on this configuration, an analog temperature sensing circuit, i.e., a voltage divider implemented using a thermistor[3] is connected to port P1. Resistances in this circuit are set to achieve $5V$ (logic 1) when temperature exceeds a fixed threshold, thus triggering a $P1$ interrupt.

Figure 13 shows the TCB implementation of $\mathcal{F}$. The TCB function is configured as an interrupt service routine for $P1$ using the $ISR(PORT1, TCB)$ compilation macro. This ensures that interrupts based on $P1$ will cause the execution of this particular function.

Once triggered, TCB disables interrupts (`dint`), calls `setup` (if this is the first TCB call after initialization/reset), and switches GPIO port P3 to active. Since P3 is connected to a buzzer[4], this turns on the alarm. The alarm remains on until P3 is set back to $0x00$. Once the alarm is on, TCB enables UART communication interface (lines 10 and 11) by setting its communication speed to a default value (constant $BAUD$) and turning it on. Next, TCB uses UART interface to send a "HELP" message (lines 13 – 20). Once sent, TCB sets port $P2$ as input, at line 22. In our sample TCB implementation, P2 is connected to an analog button used to deactivate the alarm.

[3]Thermistor is a resistance thermometer – its resistance varies with temperature.

[4]A high-frequency oscillator circuit used for generating a buzzing sound.

```
1  void setup (void) {
2     CCTL0 = CCIE;
3     CCR0  = 1000000;
4     TACTL = TASSEL_2 + MC_1;
5  }
```

Figure 14: Timer Trigger Setup

```
ISR(TIMERA0, TCB) {
   ...
}
```

Figure 15: TimerTCB Routine

Hence, TCB code goes into a while-loop that checks for an asynchronous button press, where the button is connected to the GPIO interface P2. Once a button press is detected, execution exits the while-loop and the alarm is turned off: $P3 = 0x00$. Upon completion, TCB re-enables interrupts and returns control to unprivileged software.

As discussed in Section 4, the executable corresponding to Figure 13 is also protected by *GAROTA*. Thus, its behavior cannot be modified by untrusted/compromised software.

5.2 TimerTCB: Secure Periodic Scheduling

Our second example, TimerTCB, is in the domain of real-time task scheduling. Without *GAROTA* (even in the presence of a passive RoT), a compromised MCU controlled by malware could ignore performing its periodic security- or safety-critical tasks. (Recall that targeted MCU-s typically run bare-metal software, with no OS support for preempting tasks). We show how *GAROTA* can easily ensure that a prescribed task, implemented within the TCB periodically executes.

TimerTCB only requires modifying IRQ_{cfg} to configure the timer that will cause the interruption, as illustrated in Figure 14. The *setup* function is modified to enable the MCU's built-in TIMER-A0 to cause interrupts, at line 2. Interrupts are set to occur whenever the timer's counter reaches a desired value (1 million in this example, at line 3). The timer is set to increment the counter with edges of a particular MCU clock (clock $MC1$, at line 4). As in the first example, the corresponding interrupt service routine implements the TCB (Figure 15). To define the TCB function as the interrupt handler for TIMER-A0 the compilation macro $ISR(TIMERA0, TCB)$ is used. In turn, the TCB can implement $\mathcal{F}$ as an arbitrary safety-critical periodic task.

5.3 NetTCB: Network Event-based trigger

The last example, NetTCB, uses network event-based trigger to ensure that the TCB quickly filters all received network packets to identify those that carry TCB-destined commands and take action. Incoming packets that do not contain such

```
void setup (void) {
   UART_BAUD = BAUD;
   UART_CTL  = UART_EN | UART_IEN_RX;
}
```

Figure 16: UART Trigger Setup

```
ISR(UART_RX, TCB){
   dint();
   if (first_run) {
      setup();
   }
   rxdata = UART_RXD;
   if (rxdata == 'r') {
      reset();
   }
   eint();
   return();
}
```

Figure 17: NetTCB Handler Routine and TCB Implementation

commands are ignored by the TCB and passed on to applications through the regular interface (i.e., reading from the UART buffer). In this example, we implement guaranteed receipt of external *reset* commands from some trusted remote entity. This functionality might be desirable after an MCU malfunction (e.g., due to a deadlock) is detected.

Here, trigger is configured to trap network events. IRQ_{cfg} is set such that each incoming UART message causes an interrupt, as shown in Figure 16. The TCB implementation, shown in Figure 17, filters messages based their initial character $'r'$ which is predefined as a command to *reset* the MCU. The use of the compilation macro $ISR(UART\_RX, TCB)$ causes this TCB implementation to run whenever a message is received via UART. Note that, in practice, such critical commands should be authenticated by the TCB. Although this authentication should be implemented within the TCB, we omit it from this discussion for the sake of simplicity, and refer to [24] for a discussion of authentication of external requests in this setting.

5.4 Comparison with [30, 38, 53]

As mentioned earlier in the paper, some recent work [30, 53] proposed security services that fall into the domain of active RoT-s. However, these results focus on higher-end embedded devices and require substantial hardware support: Authenticated Watchdog Timer (AWDT) implemented as a separate (stand-alone) microprocessor [53], or ARM TrustZone [30]. Each requirement is, by itself, far more expensive than the cost of a typical low-end MCU targeted in this paper (see Section 2).

In terms of functionality, both [53] and [30] are based on timers. They use AWDT to force a device reset. As in *GAROTA*, the TCB is the first to execute; this property is referred to as "gated boot" in [53]. However, unlike *GAROTA*, [30, 53] do not consider active RoT behavior stemming from other types of interrupts, e.g., as in *GAROTA* examples in Sections 5.1 and 5.3). We believe that this is partly because these designs were originally intended as an active means to enforce memory integrity, rather than a general approach to guaranteed execution of trusted tasks based on arbitrary trigger-s. Whereas, *GAROTA* is general enough to realize an active means to enforce memory integrity. This can be achieved by incorporating an integrity-ensuring function (e.g, a suitable cryptographic keyed hash) into *GAROTA* TCB and using it to check PMEM state upon a timer-based trigger.

As far as active RoT-s for lower-end MCU-s, prior work in [38] focused on enabling trusted scheduling of security-critical tasks. The architecture proposed in [38] can also enforce periodic execution of a safety-critical task that can not be interrupted, i.e., executes atomically. As such, it can be viewed as a special case of timer-based triggers in *GAROTA*. However, we note that (in addition to formal modeling and verification – see below) *GAROTA* general design differs from [38] by enabling various types of triggers, i.e., any interrupt, not just expiring timers.

Finally, we emphasize that prior results involved neither formally specified designs nor formally verified open-source implementations. As discussed in Section 1, we believe these features are important for eventual adoption of this type of an architecture.

6 Implementation & Evaluation

We prototyped *GAROTA* (adhering to its architecture in Figure 2) using an open-source implementation of the popular MSP430 MCU – openMPS430 [28] from OpenCores. In addition to *GAROTA* module, we reserve, by default, 2 KBytes of PMEM for TCB functions. This size choice is configurable at manufacturing time and MCU-s manufactured for different purposes can choose different sizes. In our prototype, 2 KBytes is a a reasonable choice, corresponding to $5-25\%$ of the typical amount of PMEM in low-end MCU-s. The prototype supports one trigger of each type: timer-based, external hardware, and network. This support is achieved by implementing the IRQ_{cfg} protection, as described in Section 4. The MCU already includes multiple timers and GPIO ports that can be selected to act as trigger-s. By default, one of each is used by our prototype. This enables the full set of types of applications discussed in Section 5.

As a proof-of-concept, we use Xilinx Vivado to synthesize our design and deploy it using the Basys3 Artix-7 FPGA board. Prototyping using FPGAs is common in both research and industry. Once a hardware design is synthesizable in an FPGA, the same design can be used to manufacture an Application-Specific Integrated Circuit (ASIC) at larger scale.

Hardware & Memory Overhead

Table 2 reports *GAROTA* hardware overhead as compared to unmodified OpenMSP430 [28]. Similar to the related work [4,15,17,19,20,42,55], we consider hardware overhead in terms of additional Look-Up Tables (LUT-s) and registers. The increase in the number of LUT-s can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate on the state overhead required by the sequential logic in *GAROTA* FSMs.

GAROTA hardware overhead is small with respect to the unmodified MCU core – it requires 2.3% and 4.8% additional LUT-s and registers, respectively. In absolute numbers, *GAROTA* adds 33 registers and 42 LUT-s to the underlying MCU.

Runtime & Memory Overhead

We observed no discernible overhead for software execution time on the *GAROTA*-enabled MCU. This is expected, since *GAROTA* introduces no new instructions or modifications to the MSP430 ISA and to the application executables. *GAROTA* hardware runs in parallel with the original MSP430 CPU. Aside from the reserved PMEM space for storing the TCB code, *GAROTA* also does not incur any memory overhead. This behavior does not depend on the number of functions or triggers used inside the TCB.

Verification Cost

We verify *GAROTA* on an Ubuntu 18.04 machine running at 3.40GHz. Results are shown in Table 2. *GAROTA* implementation verification requires checking 7 LTL statements. The overall verification pipeline (Section 4.1) is fast enough to run on a commodity desktop in quasi-real-time.

Comparison with Prior RoTs

To the best of our knowledge, *GAROTA* is the first active RoT targeting this lowest-end class of devices. Nonetheless, to provide a overhead point of reference and a comparison, we contrast *GAROTA*'s overhead with that of state-of-the-art passive RoTs in the same class. Note that the results from [30,53] can not be compared to *GAROTA* quantitatively. As noted in Section 5.4, [53] relies on a standalone additional MCU and [30] on ARM TrustZone. Both of these are (by themselves) more expensive and sophisticated than the entire MSP430 MCU (and similar low-end MCU-s in the same class). Our quantitative comparison focuses on *VRASED* [15], APEX [17], and SANCUS [42]: passive RoTs implemented on the same MCU type and thus directly comparable cost-wise. Table 3 provides a qualitative comparison among these designs. Figure 18 depicts relative overhead (in %) of *GAROTA*, *VRASED*, APEX, and SANCUS with respect to total hardware cost of the unmodified MSP430 MCU core.

In comparison with passive RoTs, *GAROTA* offers lower hardware overhead. In part, this is because, to implement its triggers, *GAROTA* leverages interrupt hardware support already present in the underlying MCU. SANCUS incurs substantially higher cost since it implements task isolation and a cryptographic hash engine (for the purpose of verifying software integrity) in hardware. *VRASED* has slightly higher cost than *GAROTA*. It includes some properties that

	Hardware		Reserved	Verification			
	Reg	LUT	PMEM/Flash (bytes)	# LTL Invariants	Verified Verilog LoC	Time (s)	Mem (MB)
OpenMSP430 [28]	692	1813	0	-	-	-	-
OpenMSP430 + *GAROTA*	725	1855	2048 (default)	7	484	3.1	13.5

Table 2: *GAROTA* Hardware overhead & verification costs.

Architecture	Behavior	Service	HW Support	Verification?
VRASED [15]	Passive	Attestation	RTL Design	Yes
SANCUS [42]	Passive	Attestation & Isolation	RTL Design	No
APEX [17]	Passive	Attestation & Proof of Execution	RTL Design	Yes
Cider [53]	Active	Timer-based trigger	Additional MCU	No
Lazarus [30]	Active	Timer-based trigger	ARM TrustZone	No
Trust. Sched. [38]	Active	Timer-based trigger	RTL Design	No
GAROTA **(this paper)**	Active	IRQ-based trigger	RTL Design	Yes

Table 3: Qualitative Comparison

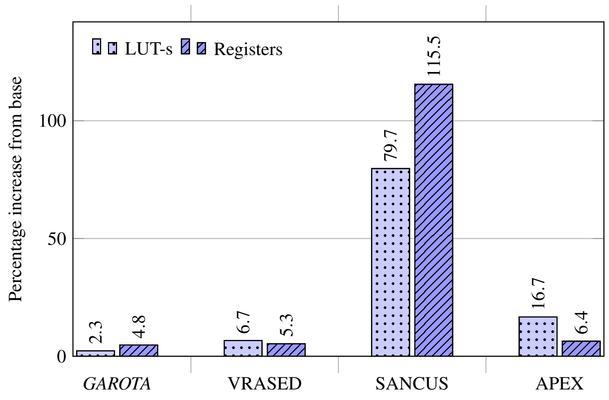


Figure 18: Comparison with passive RoTs: Hardware overhead

are similar to those of *GAROTA*, e.g., access control to particular memory segments and atomicity of its attestation implementation. Also, *VRASED* requires hardware support for an exclusive stack in DMEM. APEX hardware component is a super-set of *VRASED*, providing an additional proof-of-execution function in hardware. It thus requires strictly more hardware support, resulting in slightly higher cost. *GAROTA* also reserves approximately 3.1% (2 KBytes) of the MCU-s 16-bit address space for storing the TCB code. This value is freely configurable and is chosen as a sensible default to support envisioned RoT tasks, including sample applications in Section 5. *GAROTA*-enabled MCU-s manufactured for different use-cases could increase or decrease this amount accordingly.

7 Restrictions, Limitations & Future Directions

We now discuss some limitations and requirements imposed by *GAROTA* to the underlying MCU, as well as some directions for future work.

Remote/Runtime Software Updates. As discussed in Section 4.2.4, *GAROTA* does not affect the developer's ability to reprogram the device physically, e.g., via USB/J-TAG interface. However, if runtime software updates (e.g, sent by a remote controller) are desirable, they must be implemented within the TCB. This is because *GAROTA* prevents untrusted code from modifying program memory. In this case, the Net-TCB example (from Section 5.3) should be extended to detect "update commands" (in addition to "reset commands" in the current example). Upon receiving an "update command" the TCB would then actively monitor the UART-RX (receive) interface for further inputs from the controller and overwrite program memory as commanded. (Recall that, as discussed in Section 5.3, authenticating received commands is also crucial in this case). Untrusted applications would not be allowed to run for some time until the update is completed, as usually required by secure software updates.

Unprivileged Interrupt Disablement. By design, any unprivileged (non-TCB) software is precluded from disabling interrupts *globally* and from disabling the *GAROTA* `trigger` locally. However, disabling any other interrupt locally is still possible. Interrupts are typically disabled by default and must be actively enabled in software, e.g., as done by the `setup` function in Section 5. Any unprivileged software can keep track of currently enabled unprivileged interrupts and disable them locally as needed. For example, the maximum number of interrupts available in openMSP430 is 12. Whereas, real MSP430 deployments usually support fewer interrupt sources. Thus, we consider local disablement feasible. Nonetheless, an alternative to avoiding this burden is to implement Non-Maskable Interrupts (NMI) – which by default can not be disabled globally – and use them as *GAROTA* `trigger`-s, while allowing unprivileged applications to disable interrupts globally. However, this approach requires underlying MCU support for NMIs for each `trigger` of interest; hence it is less general.

Atomicity. As the highest priority task running on the MCU, the TCB code can not be interrupted. Any asynchronous inputs (e.g., a button press) must be actively checked by the TCB code and cannot cause nested interrupts. As discussed in Section 4, this is necessary to guarantee correct TCB execution, once triggered. We illustrate how an asynchronous input can be actively checked in the GPIO-TCB example of Section 5.1. When utilizing *GAROTA*, developers should be aware that, during TCB execution, inputs and signals that are not relevant to the TCB itself may be ignored. Generally, this is the case for any interrupt, since it is common practice in embedded system programming to disable other interrupts while an interrupt is already being serviced. Similar to regular interrupts, it is a good practice to keep TCB code short, fast, and as simple as possible.

Self-Contained TCB. In order to comply with *GAROTA* atomicity rule, the TCB code must be self-contained and entirely located within the TCB designated region in program memory. Thus, it is not possible to share common libraries with untrusted applications, since these shared libraries would need to be located outside the TCB program memory region. Therefore, all TCB libraries must be statically linked at compilation time. If the same library is used by both TCB and unprivileged software, two local copies would be needed.

Other Future Directions. As discussed in Section 2, *GAROTA* targets low-end, single-core devices that execute instructions in place from program memory (without virtual memory), and use simple memory-mapped I/O interfaces. This simplifies the specification of goals and requirements, allowing us to formally reason about *GAROTA*. From this initial step, adapting and re-designing *GAROTA* for higher-end devices, where these assumptions do no hold remains an interesting and broad open problem. Also, as discussed in Section 4.2.4, future work should consider integration of tamper-resistance and network monitoring techniques to complement *GAROTA* with security against physical attacks and adversarial network jamming (in the NetTCB case).

8 Related Work

Aside from closely related work already discussed in Section 5.4, several efforts yielded *passive* RoT designs for resource-constrained low-end devices, along with formal specifications, formal verification and provable security.

Low-end RoT-s fall into three general categories: software-based, hardware-based, or hybrid. Establishment of software-based RoT-s [26, 29, 33, 36, 48–50] rely on strong assumptions about precise timing and constant communication delays, which can be unrealistic in the IoT ecosystem. However, software-based RoTs are the only viable choice for legacy devices that have no security-relevant hardware support. Hardware-based methods [35, 39, 40, 42, 45, 47, 52] rely on security provided by dedicated hardware components (e.g., TPM [52] or ARM TrustZone [8]) or by the CPU ISA. However, the cost of such hardware is normally prohibitive for lower-end IoT devices. Hybrid RoTs [10, 15, 17, 23, 34] aim to achieve security equivalent to hardware-based mechanisms, yet with lower hardware cost. They leverage minimal hardware support while relying on software to reduce the complexity of additional hardware.

In terms of functionality, such embedded RoTs are passive. Upon receiving a request from an external trusted *Verifier*, they can generate unforgeable proofs for the state of the MCU or that certain actions were performed by the MCU. Security services implemented by passive RoTs include: (1) memory integrity verification, i.e., remote attestation [6, 10, 15, 23, 34, 42]; (2) verification of runtime properties, including control-flow and data-flow attestation [4, 17, 19, 20, 27, 40, 43, 51, 55]; as well as (3) proofs of remote software updates, memory erasure, and system-wide resets [5, 9, 16]. As discussed in Section 1 and demonstrated in Section 5, several application domains and use-cases could greatly benefit from more active RoT-s. Therefore, the key motivation for *GAROTA* is to not only provide proofs that actions have been performed (if indeed they were), but also to assure that these actions will necessarily occur.

Formalization and formal verification of RoTs for MCU-s is a topic that has recently attracted lots of attention due to the benefits discussed in Sections 1 and 4.1. *VRASED* [15] implemented the first formally verified hybrid remote attestation scheme. APEX [17] builds atop *VRASED* to implement and formally verify an architecture that enables proofs of remote execution of attested software. PURE [16] implements provably secure services for software updates, memory erasure, and system-wide resets atop *VRASED* RoT. Another recent result [11] formalized, and proved security of, a hardware-assisted mechanism to prevent leakage of secrets through time-based side-channel that can be abused by malware in control of the MCU interrupts. Inline with aforementioned work, *GAROTA* also formalizes its assumptions along with its goals and implements the first formally verified active RoT design.

9 Conclusions

This paper motivated and illustrated the design of *GAROTA*: an active RoT targeting low-end MCU-s used as platforms for embedded/IoT/CPS devices that perform safety-critical sensing and actuation tasks. We believe that *GAROTA* is the first clean-slate design of a active RoT and the first one applicable to lowest-end MCU-s, which cannot host more sophisticated security components, such as ARM TrustZone, Intel SGX or TPM-s. We believe that this work is also the first formal treatment of the matter and the first active RoT to support a wide range of RoT trigger-s.

Acknowledgments

The authors sincerely thank USENIX Security'22 reviewers. This work was supported by funding from the Semiconductor Research Corporation (SRC) Contract 2019-TS-2907, NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs. Part of this work was conducted while the second author was at the University of California, Irvine.

References

[1] 2020 CISCO global networking trends report. https://www.cisco.com/c/dam/m/en_us/solutions/enterprise-networks/networking-report/files/GLBL-ENG_NB-06_0_NA_RPT_PDF_MOFU-no-NetworkingTrendsReport-NB_rpten018612_5.pdf, 2020.

[2] *GAROTA* source code. https://github.com/sprout-uci/garota, 2022.

[3] Tigist Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. Things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[4] Tigist Abera et al. C-flat: Control-flow attestation for embedded systems software. In *CCS '16*, 2016.

[5] Mahmoud Ammar and Bruno Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *Annual Computer Security Applications Conference*, pages 717–732, 2020.

[6] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[7] Emekcan Aras, Mahmoud Ammar, Fan Yang, Wouter Joosen, and Danny Hughes. Microvault: Reliable storage unit for iot devices. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 132–140. IEEE, 2020.

[8] Arm Ltd. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone, 2018.

[9] N Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018.

[10] F. Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *DAC*, 2015.

[11] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 262–276. IEEE, 2020.

[12] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[13] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[15] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security*, 2019.

[16] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems. 2019.

[17] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association.

[18] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the toctou problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[19] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[20] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 24. ACM, 2017.

[21] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata

manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, 2016.

[22] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Wisec*, 2017.

[23] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, 2012.

[24] Brasser et al. Remote attestation for low-end embedded devices: the prover's perspective. In *DAC*, 2016.

[25] Robert Faludi. *Building wireless sensor networks: with ZigBee, XBee, arduino, and processing*. " O'Reilly Media, Inc.", 2010.

[26] Ryan W Gardner, Sujata Garera, and Aviel D Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE TIFS*, 2009.

[27] Munir Geden and Kasper Rasmussen. Hardware-assisted remote runtime attestation for critical embedded systems. In *2019 17th International Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2019.

[28] Olivier Girard. openMSP430, 2009.

[29] Virgil D Gligor and Shan Leung Maverick Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[30] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The lazarus effect: Healing compromised devices in the internet of small things. *arXiv preprint arXiv:2005.09714*, 2020.

[31] Intel. Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[32] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A tool for word-level verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[33] Rick Kennell and Leah H Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, 2003.

[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*, 2014.

[35] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P '12*, 2012.

[36] Yanlin Li, Jonathan M McCune, and Adrian Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS*, 2011.

[37] Marwa Mamdouh, Mohamed AI Elrukhsi, and Ahmed Khattab. Securing the internet of things and wireless sensor networks via machine learning: A survey. In *2018 International Conference on Computer and Applications (ICCA)*, pages 215–218. IEEE, 2018.

[38] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 61–70, 2012.

[39] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P '10*, 2010.

[40] Jonathan McCune et al. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 2008.

[41] Kenneth L McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[42] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, et al. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3), 2017.

[43] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 641–646. IEEE, 2021.

[44] Johannes Obermaier and Vincent Immler. The past, present, and future of physical security enclosures: from battery-backed monitoring to puf-based inherent security and beyond. *Journal of Hardware and Systems Security*, 2(4):289–296, 2018.

[45] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[46] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.

[47] Dries Schellekens et al. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 2008.

[48] A. Seshadri et al. SWATT: Software-based attestation for embedded devices. In *IEEE S&P '04*, 2004.

[49] A. Seshadri et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[50] Arvind Seshadri et al. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

[51] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[52] Trusted Computing Group. Trusted platform module (tpm), 2017.

[53] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1415–1430. IEEE, 2019.

[54] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. A survey of intrusion detection in internet of things. *Journal of Network and Computer Applications*, 84:25–37, 2017.

[55] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.

[56] Wu Zhijun, Li Wenjing, Liu Liang, and Yue Meng. Low-rate dos attacks, detection, defense, and challenges: a survey. *IEEE Access*, 8:43920–43943, 2020.

[57] Wu Zhijun, Li Wenjing, Liu Liang, and Yue Meng. Low-rate dos attacks, detection, defense, and challenges: a survey. *IEEE Access*, 8:43920–43943, 2020.