Resolving Availability and Run-time Integrity Conflicts in Real-Time Embedded Systems

Adam Ilyas Caulfield
University of Waterloo
Waterloo, Canada
acaulfield@uwaterloo.ca

Muhammad Wasif Kamran
University of Waterloo
Waterloo, Canada
mwkamran@uwaterloo.ca

N. Asokan
University of Waterloo
Waterloo, Canada
asokan@acm.org

Abstract

Run-time integrity enforcement in real-time systems presents a fundamental conflict with availability. Existing approaches in real-time systems primarily focus on minimizing the execution-time overhead of monitoring. After a violation is detected, prior works face a trade-off: (1) prioritize availability and allow a compromised system to continue to ensure applications meet their deadlines, or (2) prioritize security by generating a fault to abort all execution.

In this work, we propose PAIR, an approach that offers a middle ground between the stark extremes of this trade-off. PAIR monitors real-time tasks for run-time integrity violations and maintains an *Availability Region (AR)* of all tasks that are *safe to continue*. When a task causes a violation, PAIR triggers a non-maskable interrupt to kill the task and continue executing a non-violating task within AR. Thus, PAIR ensures only violating tasks are prevented from execution, while granting availability to remaining tasks. With its hardware approach, PAIR does not cause any run-time overhead to the executing tasks, integrates with real-time operating systems (RTOSs), and is affordable to low-end microcontroller units (MCUs) by incurring +2.3% overhead in memory and hardware usage.

1 Introduction

Embedded systems have become ubiquitous, often serving important roles in critical infrastructure responsible for real-time sensing or actuation, where deterministic and timely responses are crucial for functionality and safety. They are implemented with microcontroller units (MCUs), low-powered resource-constrained processing units optimized for energy efficiency. MCUs typically lack advanced features, such as memory management units (MMUs) or the ability to run rich operating systems (OS) that facilitate inter-process isolation. To make up for this, real-time operating systems (RTOSs) can be incorporated as an additional software layer to provide basic task management (e.g., scheduling, inter-task communication, timing services) with low overhead. Still, limited architectural protections make MCUs vulnerable [29].

Methods like secure boot [25] ensure the code integrity at boot-time, but additional measures are required to provide integrity guarantees at run-time. For example, an adversary ($\mathcal{A}$dv) could inject code into memory, then exploit a vulnerability (e.g., a buffer overflow [17]) to redirect execution to the injected code. Although these code-injection-based control flow attacks can be mitigated with memory protection techniques (e.g., $W \oplus X$ and DEP [26]), they do not protect against code-reuse attacks like return- or jump-oriented programming (ROP/JOP) [9, 39], that chain together carefully selected instruction gadgets from unmodified code. To account for such attacks, run-time integrity mechanisms are required.

A prominent run-time integrity mechanism is Control Flow Integrity (CFI) [10], which employs reference monitors to detect when control flow instructions (e.g., calls, returns, jumps) deviate from an expected control flow path. Typically, valid paths are determined by constructing a Control Flow Graph (CFG) via static analysis and comparing the destination of a control flow transition with the set of valid locations in the CFG. Since the exact destination of some branch instructions (e.g., returns, indirect jumps/calls) cannot be determined from static analysis, mechanisms like shadow stacks [11], type-checking [28], and landing pads or pointer authentication [8] can be used to narrow the scope of valid destinations.

Recent CFI methods for embedded and real-time systems [27] enable cost-effective enforcement of valid control flows. Several designs [2, 15, 23, 40, 45, 46] focus on performing CFI checks in a way that minimizes run-time overhead on a program (or *task*) while monitoring integrity. However, most works overlook the handling of tasks *after* a CFI violation has been raised. On the other hand, availability-focused security techniques do not account for run-time violations [4, 18], delay intervention until task completion [2], or invoke the scheduler without task integrity guarantees [3]. Given this trade-off, it is unclear from current works how to achieve a middle ground between post-violation integrity and availability.

To bridge this gap, we introduce PAIR: ***P**reserving **A**vailability and **I**ntegrity at **R**un-time*. PAIR demonstrates that both availability and run-time integrity can be prioritized simultaneously. PAIR hardware components monitor run-time integrity of independent tasks within a given program. Then, PAIR dynamically adds and removes them from an Availability Region (AR) based on their integrity status. When any run-time integrity violation is detected, PAIR aborts execution of the violating task, removes it from AR, and continues executing another task in AR. PAIR does not allow a task to return to AR until a trusted software update occurs. To summarize, we claim the following contributions:

- PAIR, an architecture operating alongside an RTOS that maintains availability and integrity of benign tasks after integrity violations due to other tasks (Sec. 3);
- formally defined properties of PAIR security, its modules sub-properties, and demonstration via model-checking that the sup-properties and security goals are upheld (Sec. 4);
- design and implementation of PAIR hardware submodules as formally-verified finite state machines (FSMs) that implement the outlined sub-properties (Sec. 6);
- an open-source FPGA prototype of PAIR [5], evaluated with a real-world RTOS and example benchmark applications, demonstrating applicability to low-end MCUs due to minimal memory and hardware overheads (Sec. 7).

2 Background

2.1 Scope

We consider embedded systems operating real-time or sensing-based tasks atop low-end, single-core MCUs, with limited addressable memory (e.g., 8-256KB), split into program memory (PMEM) and data memory (DMEM). They are typically 8- or 16-bit CPUs running at 1-16MHz (e.g., TI MSP430). Such devices usually run software at *bare metal*, meaning they lack MMUs, and thus perform memory accesses and execute instructions from their physical addresses. We assume an RTOS is implemented for task scheduling and timing services.

2.2 Control Flow Integrity (CFI)

CFI is a countermeasure for control flow hijacking attacks, which exploit memory vulnerabilities to launch unintended run-time behavior by overwriting *control data* (e.g., return addresses, indirect jump targets, function pointers). CFI defends against such attacks by restricting the targets of control flow instructions (e.g., returns, calls, jumps) to a set of predetermined valid destinations. It requires two components: (1) valid destination sets for each control flow instruction (at either coarse- or fine-granularity), and (2) a reference monitor to detect violations and respond accordingly.

Many techniques employ their reference monitor through software instrumentation [38, 43, 49, 50], enabling deployment of CFI without hardware changes. Alternatively, hardware extensions that enable CFI, such as Branch Target Identification (BTI) and Pointer Authentication (PA) in ARM [8] and Intel Control-Flow Enforcement Technology (CET) [42] in commercial processors, have led to subsequent proposals for fully fledged CFI schemes [6, 21, 24, 48]. ARM has made BTI and PA available among their class of embedded devices, along with hardware tracing extensions that have also been leveraged to deploy CFI in embedded systems [41, 46].

2.3 Real-Time Operations in Embedded Systems

Embedded systems can implement an RTOS that efficiently provides critical services for real-time operations (e.g., task scheduling, inter-task communication, timing adherence). After an application assigns tasks priority and schedules them through an RTOS API, the RTOS manages their execution. To do so, the RTOS contains functions to control execution, such as killing a task or yielding to another task. A task runs until it voluntarily yields or until a time period has elapsed, at which point the RTOS intervenes to perform a context switch and resume the next highest-priority task.

Many works have proposed security mechanisms that account for real-time operations in embedded systems, including mechanisms to provide authenticated [13, 30–32] and audited [12, 14] proofs of valid execution, availability mechanisms for trusted software [3, 4, 18], and local run-time integrity in the form of CFI [27]. CFI for embedded systems is either fully software-based [2, 40, 45] or uses hardware extensions with minimal software support [23, 41, 46]. Regardless of the approach, all aim to minimize disruptions to the executing task during the CFI monitoring. Upon the violation, they abort execution to prioritize run-time integrity, compromising the availability of a system in an unknown state. However, in the case of real-time systems, aborting execution system-wide can be costly. An alternative approach [2] is to allow the system to log the violation and continue executing. Although this prioritizes availability, it could be costly to allow a system in an unknown state to continue execution.

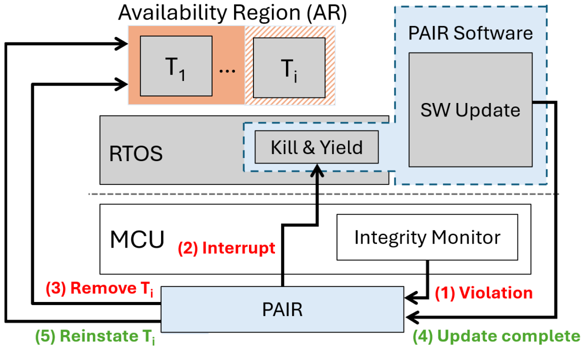

Figure 1: PAIR Overview: Tasks and an RTOS execute in software, with all tasks initially in the AR. After detecting an integrity violation (1), PAIR triggers the trampoline into RTOS to *kill-and-yield* (2) and removes the violating task from AR (3). Only after a trusted software update has completed (4) will PAIR reinstate the violating task into AR (5).

2.4 Linear Temporal Logic

We use Linear Temporal Logic (LTL) [44] to specify a formal model of PAIR hardware and the properties that it should satisfy. In Sec. 7, we use a model checking (specifically NuSMV [16]) to determine whether the system adheres to the specified properties. Similarly to prior works [4, 13, 34? , 35], we describe hardware using LTL specifications containing propositional connectives (e.g., conjunctions $\wedge$, disjunctions $\vee$, negation $\neg$, implication $\rightarrow$) and temporal connectives (e.g., $X\phi$ to denote a condition ϕ is true at the next system state , and $G\phi$ to denote a condition ϕ globally holds for all future states) between system states.

3 PAIR at a High Level

Unlike a typical system in which a run-time integrity monitor (IM) would directly invoke a system reset signal, PAIR's goal is to ensure non-violating tasks can continue safely after a run-time integrity violation has been detected. Therefore, upon receiving a redirected violation signal, PAIR can perform operations to ensure benign tasks can securely continue. PAIR goals can be reduced to upholding the following two security properties at run-time:

[P1] Post-Violation Availability: when a task has a run-time integrity violation, PAIR always interferes to (1) remove the task from AR and (2) yield to another non-violating task;

[P2] Post-Violation Integrity: tasks outside AR can never re-execute until a software update has occurred.

A high-level overview of an PAIR-enabled system is depicted in Fig. 1. It assumes a system that operates real-time tasks (denoted T_i) with an RTOS, while PAIR introduces two trusted software components. Initially, the MCU runs with no violations, and PAIR has assigned all tasks to AR. PAIR monitors MCU signals and memory to determine which real-time task is executing at any given moment. PAIR interfaces with an IM to detect the violation when it occurs (1). Upon this detection, PAIR will generate a non-maskable interrupt (2) to invoke the *kill-and-yield* trampoline, while also removing the violating task from AR. These steps are repeated for any other tasks that generate a violation. The bounds of AR are not expanded again until a trusted software update has completed (4), and at this point PAIR will reinstate all violating tasks back into AR.

4 Adversary & System Models

4.1 Adversary Model

We consider an adversary ($\mathcal{A}$dv) that exploits software vulnerabilities to manipulate any software or data, unless it is explicitly protected by hardware. $\mathcal{A}$dv aims to cause run-time integrity violations (e.g., control flow hijack and ROP/JOP attacks [9, 39]) to invoke malicious actions or avoid critical actions. $\mathcal{A}$dv aims to continue its malicious actions even after detection, hoping to prevent intervention or other tasks from executing. Hardware attacks that require physical attacks to circumvent hardware protections are out of scope for this work, as they require orthogonal physical access control measures [36].

4.2 Hardware and Memory Configuration

Fig. 2 shows the assumed initial hardware configuration and memory layout. PAIR interfaces with the MCU to read the following signals pertaining to its run-time behavior:

- *PC*: the program counter register storing the memory address of the instruction that is executing;
- W_{en}: a flag denoting whether the instruction is performing a write to memory;
- R_{en}: a flag denoting whether the instruction is performing a read from memory;
- D_{addr}: the memory address that is targeted by the current instruction (i.e., the address being written to/read from if W_{en}/R_{en});
- *irq*: MCU signal that indicates whether an interrupt is happening (i.e., the MCU has identified the interrupt source and is currently interrupting the current execution).

We assume the MCU is extended with an existing IM that interfaces with relevant MCU signals. We assume the IM outputs a signal $violation_{IM}$ that is only set when the current instruction is violating its security policy. This signal is passed into PAIR hardware.

MCU memory is divided into program memory (PMEM) and data memory (DMEM). PMEM contains fixed sub-regions for the following components:

- TR: the task region containing all real-time tasks;
- RTOS: the region containing the RTOS itself;
- SW: the region containing PAIR trusted software.

DMEM contains the following fixed sub-regions:

- D: memory used for tasks and RTOS;
- D$_{\text{PAIR}}$: contains all critical data referenced by SW or hardware.

D$_{\text{PAIR}}$ contains a *key* used by SW trusted update function, the number of tasks configured in TR (*N*), and their bounds within TR (i.e., for each T_i, the task bounds are specified as a T_i^{min} and T_i^{max} holding the minimum and maximum PMEM addresses pertaining to T_i). We assume that task bounds are non-overlapping and independent in terms of their code, but may share data. PAIR hardware monitors signals from MCU and IM to monitor tasks and their availability (i.e., maintain AR), and outputs a *trigger*, which implements a non-maskable interrupt to invoke SW.

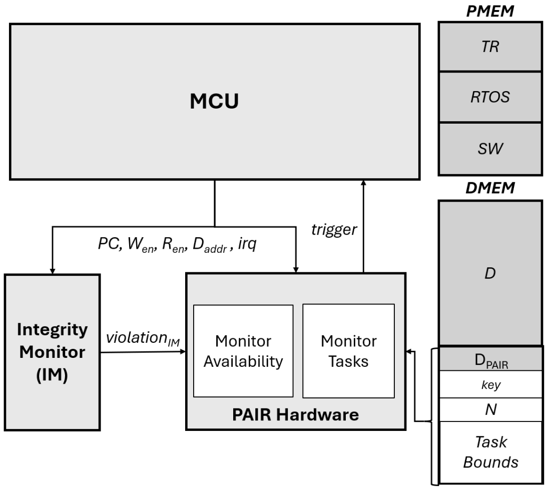

Figure 2: System Configuration: PAIR interfaces with the MCU to read specific signals pertaining to the currently executing instruction (e.g., PC, W_{en}, R_{en}, D_{addr}, irq**).**

4.3 Formalized Security Properties

We formalize the security properties of PAIR in Fig. 3. We define **[P1]** as the two LTL statements in Definition 1. Given a run-time integrity violation is occurring ($violation_{IM}$ is true) or another access-control violation to PAIR critical components ($violation_{PAIR}$ is true), PAIR's kill-and-yield trigger (*trigger*) should be set in the next state. Given task *i* (T_i) is executing (i.e., *PC* is within the bounds of TR and the current $task_{id}$ is *i*) and PAIR has generated *trigger*, the violating task T_i should no longer be in AR in the next state. We define **[P2]** in Definition 2. If T_i is executing, it must also be true that either T_i is in AR or that *trigger* will be set in the next state.

5 Security Sub-properties and Proofs

5.1 PAIR module Sub-properties

Based on PAIR's goals, we outline a set of required sub-properties to be implemented by PAIR's design. These sub-properties fall into two classes: task monitoring and availability monitoring. Later in Sec. 5.2, we demonstrate that these sub-properties suffice to achieve PAIR's desired security properties.

An axiom as a part of the task monitoring specification is the interaction with D$_{\text{PAIR}}$ to read the task boundaries. This is outlined in Fig. 4. Given a fixed set of tasks *N* with their bounds installed in fixed locations, we assume LTL 1 that specifies the configuration of an internal label $task_{id}$. This label is set depending on which task bounds *PC* is currently within. Given *PC* is within the bounds of T_i, $task_{id}$ is set to *i*.

Two additional sub-properties are required to ensure that tasks do not perform any illegal memory accesses. These sub-properties are outlined in Fig. 5. The sub-property specified in LTL 2 requires

Definition 1: Post-Violation Availability

$$G : \{violation_{\mathrm{PAIR}} \lor violation_{IM} \rightarrow X(trigger))\}$$

$$G : \{(PC \in TR) \land (task_{id} = i) \land trigger \rightarrow \neg X(T_i \in AR)\}$$

Definition 2: Post-Violation Integrity

$$G : \{(PC \in TR) \land (task_{id} = i) \rightarrow (T_i \in AR) \lor X(trigger)\}$$

Figure 3: Formal Specification of PAIR security properties

Tracked Task Execution:

$$G : \{PC \leq T_i^{max} \land PC \geq T_i^{min} \rightarrow (task_{id} = i)\} \quad \forall i \in \{0, N-1\} \tag{1}$$

Figure 4: Axiom: Tracking currently executing task.

Task Memory Access Integrity:

$$G : ((R_{en} \lor W_{en}) \land (D_{addr} \in D_{\mathrm{PAIR}}) \land \neg(PC \in SW)) \rightarrow violation_{\mathrm{PAIR}}) \tag{2}$$

$$G : (W_{en} \land (D_{addr} \in PMEM) \land \neg(PC \in SW) \rightarrow violation_{\mathrm{PAIR}}) \tag{3}$$

$$G : (violation_{\mathrm{PAIR}} \rightarrow X(trigger)) \tag{4}$$

Figure 5: Sub-properties to monitor task execution.

Definitions: N-bit bitmasks.

$$\mathtt{mask}(id) := (1 << id), zero := 0_N, set := 1_N$$

Triggering upon integrity monitor violation:

$$G : \{violation_{IM} \rightarrow X(trigger)\} \tag{5}$$

$$G : \{trigger \land irq \rightarrow \neg X(trigger)\} \tag{6}$$

Reducing AR coverage upon trigger:

$$G : \{((PC \in TR) \land (task_{id} = i)) \land trigger \rightarrow (X(AR_{en}) \ \& \ \mathtt{mask}(task_{id})) = zero)\} \tag{7}$$

Triggering upon re-execution of violating task:

$$G : \{((PC \in TR) \land (task_{id} = i)) \land (AR_{en} \ \& \ \mathtt{mask}(task_{id})) = zero) \rightarrow X(trigger)\} \tag{8}$$

Resetting ***AR*** **after software update:**

$$G : \{PC = SW_{exit} \rightarrow X(AR_{en}) = set\} \tag{9}$$

Figure 6: Sub-properties to maintain AR.

that PAIR trigger SW to remove task i from AR when that task attempts to read or write to D_{PAIR} while outside of SW. This is defined by setting $violation_{PAIR}$ when a read or write instruction is targeting D_{PAIR} while not executing within SW. The second sub-property specifies that $violation_{PAIR}$ should be set when software that is not in SW attempts to modify PMEM, as defined in LTL 3. Whenever $violation_{PAIR}$ is set, $trigger$ must be set in the next state, as specified in LTL 4.

Note: *Further access control is possible if implemented in the IM. For example, our prototype IM (described in Sec. 7) enforces CFI and prevents tasks from modifying each other's stacks.*

The required sub-properties for monitoring availability and maintaining AR are outlined in Fig. 6. PAIR maintains a bit vector AR_{en} that signifies whether a task is a part of the AR: removing or adding a task to AR corresponds to clearing or setting its corresponding bit in AR_{en}. Therefore, changes to AR are formally specified by bitwise operations that clear or set corresponding bits in AR_{en}. To model this, we define mask(id) as a method to map an id to an N-length bitmask. Similarly, we define $zero$ and set as N-length bitmasks of all zeros and ones, respectively.

Theorem 1: LTLs 2, 3, 4, 5- 6 ⟶ Definition 1

Theorem 2: LTLs 1, 7, 8, 9 ⟶ Definition 2

Figure 7: Theorems that [P1] and [P2] are upheld.

LTL 5 and 6 specify the requirements to control PAIR's non-maskable interrupt signal *trigger*. LTL 5 shows that *trigger* should be set in the next state when $violation_{IM}$ has been set by the IM. LTL 6 shows that *trigger* is cleared in the next state when *trigger* is set in the current and the *irq* has been set, denoting that the interrupt has been accepted by the MCU.

LTL 7 requires that the bit in AR_{en} corresponding to the executing task is cleared when *trigger* has been generated. This is specified by requiring that the bitwise AND of mask($task_{id}$) and AR_{en} in the next state equals zero.

Any attempts to re-execute a task that has been removed from AR should also result in the invocation of PAIR's software to yield to another non-violating task. This is specified by LTL 8 by requiring that *trigger* is set in the next state when the bit in AR_{en} corresponding to T_i has been cleared. Finally, AR should only be reset (i.e., all bits in AR_{en} set to one) when exiting from SW. This is specified in LTL 9.

5.2 PAIR goals via sub-properties

Fig. 7 outlines the theorems that state properties **[P1]** and **[P2]** are upheld. These theorems are also checked via model checker, described further in Sec. 7. Here we also discuss the intuition behind why these theorems uphold the security properties.

Theorem 1 states that LTLs 2, 3, 4, 5, and 6 uphold Definition 1 for **[P1]**. Definition 1 specifies that each instance of $violation_{IM}$ or $violation_{PAIR}$ must always result in *trigger* in the next state. It is provided by LTLs 5 and 6 that *trigger* is set at this instance and is not cleared until the MCU generates its interrupt signal, denoting it was accepted. Additionally, LTLs 2 and 3 ensure that a task's memory accesses that are relevant to PAIR result in *trigger*, regardless of the specific IM in use.

Theorem 2 states that LTLs 1, 7, 8, 9 uphold Definition 2 for **[P2]**. Definition 2 describes post-violation integrity by specifying that execution of a task in a given state implies either that the task is in AR or that *trigger* will be generated in the next state. This property requires monitoring of the currently executing task, which is specified in LTL 1 to be maintained in $task_{id}$. LTL 7 specifies that given *trigger* has been generated, the current task is removed from AR by performing the corresponding bitwise operations based on $task_{id}$. Additionally, LTL 8 supports Definition 2 by ensuring any attempted re-entering of a removed task results in *trigger* generation, specified by generating *trigger* when AR_{en} shows the current task was revoked from AR. Finally, LTL 9 also supports Definition 2 by specifying tasks can be re-added to AR after exiting from SW, and thus they will not generate *trigger* if they have been re-executed after a software update.

6 Implementation & Verification

PAIR modules are implemented as finite state machines (FSM) in Verilog. Here we specify two finite state machines: one for generating *trigger*, and a second for controlling AR_{en}.

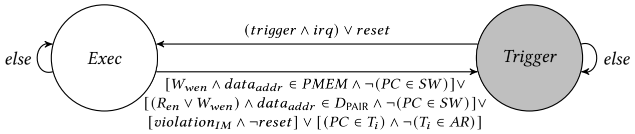
Exec
Trigger
else
else
$(trigger \wedge irq) \vee reset$
$[W_{wen} \wedge data_{addr} \in PMEM \wedge \neg(PC \in SW)] \vee$
$[(R_{en} \vee W_{wen}) \wedge data_{addr} \in D_{\text{PAIR}} \wedge \neg(PC \in SW)] \vee$
$[violation_{IM} \wedge \neg reset] \vee [(PC \in T_i) \wedge \neg(T_i \in AR)]$

Figure 8: Verified FSM for LTLs 2, 3, 5, 6.

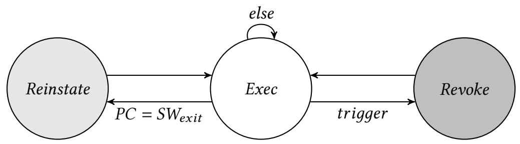

Figure 9: Verified FSM for LTLs 7, 9.

6.1 Sub-module FSMs

Fig. 8 shows the FSM for generating and clearing *trigger*. The value of *trigger* is set based on either of the following states:

- *Exec*: representing the state in which the MCU is executing without a violation, hence *trigger* is unset;
- *Trigger*: representing the state in which a violation has occurred and *trigger* is set.

The MCU enters *Exec* upon any reset, and thus this is the initial state. A transition from *Trigger* into *Exec* occurs based on the specification in LTL statements 2, 3, and 5 corresponding to run-time violations. A transition from *Trigger* into *Exec* only occurs when the *trigger*-based interrupt has been accepted by the MCU, corresponding to LTL statement 6.

Fig. 9 shows the FSM for modifying AR via its corresponding bitvector AR_{en}. Each state corresponds to the status of AR_{en}:

- *Exec*: state corresponding to execution that does not require reinstating or revoking a task to/from AR, and thus AR_{en} remains unchanged in this state;
- *Revoke*: state corresponding to that in which a task must be removed from AR, and the corresponding bit of the violating task is cleared in this state;
- *Reinstate*: state corresponds to a successful software update and thus all tasks are reinstated in AR.

This FSM initializes in the *Exec* state. Only upon a trigger does a transition into *Revoke* occur, and a transition from *Exec* into *Reinstate* occur only occurs when a software update completes (i.e., PC being equal to SW_{exit}). Both *Revoke* and *Reinstate* states only execute for one cycle to update AR accordingly before transitioning to *Exec*. The transition from *Exec* into *Revoke* corresponds to LTL 7, and the transition from *Exec* into *Reinstate* corresponds to LTL 9.

6.2 Formal Verification

We use NuSMV [16] as a model checker to determine whether PAIR design adheres to the specified properties and statements from Sec. 5 are upheld. We use Verilog2SMV [**?**] to convert Verilog specifications into nuSMV statements. Then, we formulate the LTL specifications in nuSMV to check the model. We execute the model checking atop an Intel(R) Core(TM) Ultra 7 155H (3.80 GHz) with 32.0 GB RAM using Ubuntu-20.04 under Windows Subsystem for Linux 2 (WSL 2). The model checker executed for 43.98 seconds with peak memory usage of 1.38GB to complete the verification.

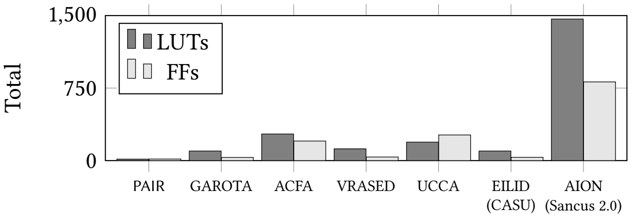

Figure 10: Hardware cost of PAIR is negligible and significantly lower than related works.

7 Prototype & Evaluation

7.1 Prototype configuration

To implement PAIR prototype, we use the Xilinx Vivado tool-set [47] as a development platform. PAIR hardware is written in the Verilog hardware description language and implements each of PAIR sub-modules according to the logic defined in Sec. 6, and its software is implemented in C. Using Vivado, we synthesize PAIR hardware, simulate behavior, evaluate area/energy costs, and deploy a PAIR-equipped openMSP430 [22] on the Basys3 prototyping board [19] We configure openMSP430 with 16KB of PMEM and 12KB of DMEM.

A custom linker is used to ensure all code and data of interest (from Fig. 2) are placed into contiguous memory regions that are monitored by PAIR. For the RTOS in our evaluation, we use RIOT [1] due to its open-source availability and support for MCUs within our scope. In theory, PAIR concepts can apply to any RTOS with an identifiable API for *kill* and *yield* procedures.

For the IM, we instantiate a task-aware shadow stack module along with an indirect-branch table (IBT) for each task in hardware. Both interface with MCU memory to store their data and monitor MCU signals as required. It generates a violation upon illegal return instructions according to the shadow stack module and illegal forward branch instructions according to the indirect branch tables. Additionally, it monitors tasks' stacks to ensure they do not interfere with each other's temporary variables.

7.2 Fixed Costs

PAIR hardware cost is accounted for in the Look-up tables (LUTs) and flip-flop registers (FFs) added to the openMSP430 core. We compare this cost to that of closely related works in Fig. 10.

PAIR requires fewer resources than all closely related works that deploy security extensions atop openMSP430. Although PAIR has a different goal than each of these works, this shows that it is cost-effective for low-end devices and is in line with related works that target the same class of devices. The most closely related works are EILID [23] and AION [3]. EILID implements CFI for low-end MCUs atop a hybrid (software-hardware) root of trust CASU [18] for memory immutability and trusted software updates. We note that when including IM in the cost, PAIR + IM requires more hardware resources than EILID. However, we note that PAIR is concerned with providing availability and run-time integrity simultaneously, whereas EILID's primary focus is run-time integrity (discussed further in Sec. 8). AION builds atop Sancus [33] hardware-based TEE, which is more expensive due to requiring full TEE features and incorporating a cryptographic engine.

		Usage in bytes	
Application	Attribute	PAIR	PAIR + IM
sched_round_robin	MT	12 (+0.073%)	24 (+0.146%)
ipc_pingpoing	MT	12 (+0.073%)	14 (+0.085%)
thread_priority_inversion	MT	16 (+0.097%)	22 (+0.134%)
mutex_sleep	MT	8 (+0.048%)	12 (+0.073%)
sched_change_priority	MT	8 (+0.048%)	12 (+0.073%)
crc_32	SUI	4 (+0.024%)	8 (+0.048%)
matrixmult	SUI	4 (+0.024%)	12 (+0.073%)
lcdnum	IF, SUI	4 (+0.024%)	42 (+0.256%)
cover	IF, SUI	4 (+0.024%)	374 (+2.28%)
cubic	SUI	4 (+0.024%)	10 (+0.061%)
fdct	SUI	4 (+0.024%)	10 (+0.061%)
tarai	R	4 (+0.024%)	18 (+0.109%)
recursion	R	4 (+0.024%)	26 (+0.159%)

Table 1: Memory usage (bytes) for applications with multiple tasks (MT), recursion (R), indirect forward edge control flows (IF), and high or very high suitability for MCUs (SUI) [37].

We also evaluate PAIR performance overheads by considering additional energy consumption. To evaluate energy overheads, we use Vivado synthesis tools to estimate PAIR's power consumption on the Basys3 FPGA board. We find the openMSP430 core without PAIR consumes 71 mW of static power. Performing the same analysis on PAIR-equipped openMSP430, we observe 76 mW of static power, showing that PAIR incurs only an additional 5 mW in static power consumption.

7.3 Performance costs

We evaluate the performance costs of PAIR by evaluating the run-time and memory overheads when executing software atop PAIR. PAIR hardware does not impose any run-time overheads, since it is monitoring MCU signals without affecting the MCU's critical path. The memory cost incurred by PAIR varies per application. For instance, D_{PAIR} size depends on the number of independent tasks. Additionally, since the IM may have a memory footprint, additional memory overhead is incurred while using PAIR that is not due to PAIR. Our IM instance memory usage for shadow stacks depends on the maximum call depth among all tasks, and IBT size depends on the total number of indirect forward edge control flow transfers among all tasks. To effectively evaluate PAIR memory usage, we select a set of applications from RIOT test examples and BEEBs embedded systems benchmark suite [37]. We select RIOT test examples that create multiple tasks, and BEEBS applications that meet one or more of the following criteria:

(1) has one or more recursive calls;
(2) has one or more indirect forward edges (jump/call);
(3) has *high* or *very high* suitability for MCUs [37].

Table 1 shows the resulting memory usage due to PAIR alongside our IM instantiation.

Among the selected applications, PAIR memory usage is minimal, an increase ranging from just 4 to 16 bytes (0.024% ot 0.097% of available DMEM). Additional memory usage is required due to the IM, but it is still minimal: an increase ranging from 8 to 42 bytes (0.048% to 2.28% of the available data memory). PAIR by itself has a minimal fixed memory cost ($2 + 2 \times N$ bytes), but incurs additional variable costs due to the IM that is installed.

8 Related Work

Many works have explored CFI in embedded and real-time systems [27]. InsectACIDE [46] proposes a method for CFI for ARM Cortex-M MCUs leveraging debugging extensions [7] to record control flow transfers without incurring run-time overhead. Then, recorded control flow transfers are checked for integrity violations during idle times. RECFISH [45] demonstrates an RTOS-compatible CFI method for ARM Cortex-R MCUs by implementing shadow stacks and function-label checks via software instrumentation. FastCFI [20] also uses ARM extensions, but instead uses an FPGA to store and check a CFG. Regardless of their specificities, PAIR differs from each in that it accounts for the post-violation phase, allowing non-violating tasks to continue their execution.

EILID [23] proposes a hybrid method for low-end MCUs and also evaluates atop openMSP430. It uses code instrumentation to implement trampolines that jump to a hardware-protected CFI monitor. PAIR +IM incurs more hardware overhead, while EILID incurs more run-time overhead and an increase in code size. Similar to PAIR, EILID triggers an interrupt upon violations and also enables software updates. However, EILID's interrupt jumps directly to the software update function. PAIR's interrupt instead disables a violating task before returning to the next high-priority benign task. Software updates may occur independently upon receiving an authenticated message. Due to parallel goals, PAIR and EILID could coexist, using PAIR to provide post-violation availability/integrity while using EILID to obtain a low-cost IM.

Several related works address availability guarantees under full software compromise in MCUs [3, 4, 12, 14]. Among them is AION [3], which builds upon Sancus TEE [33] to provide strong availability guarantees to a trusted scheduler in the presence of compromised software. Like AION, we also built our prototype using openMSP430 as an MCU and RIOT as an RTOS. AION ensures the availability of a trusted Sancus-aware scheduler by incorporating hardware-based atomicity monitor and exception engine modules. While both AION and PAIR invoke the RTOS scheduler upon a violation signal, they pursue complementary objectives. AION aims to preserve the availability of a TEE-protected trusted scheduler to fairly allocate execution time to tasks, whereas PAIR focuses on preventing tasks that violate run-time integrity policies from executing until they are updated, thereby preserving availability to other non-violating tasks on the same platform. This is provided even if other software, including the scheduler, attempts to re-enter it. Given the orthogonal goals and commonality between MCU and RTOS platform for prototyping, PAIR and AION combined could provide a strong security service: AION's trusted scheduler availability with PAIR's enforcement against re-entering of violating tasks could return an architecture that ensures fair/timely scheduling and post-violation containment.

Active roots of trust (ARoTs) [4, 12, 14], first proposed by Nunes et al. [4], ensure a contiguous region of memory (housing a trusted interrupt service routine) is guaranteed to execute upon its corresponding signal or upon MCU reset. Unlike ARoTs, which monitor a statically defined PMEM region, PAIR dynamically controls which tasks are granted availability.

9 Conclusion

In this work, we propose PAIR to preserve task availability and system-wide run-time integrity. By ensuring that only the offending task is prevented from executing, PAIR enables non-offending critical real-time tasks to continue executing. We designed, implemented, and evaluated a formally verified and open-source prototype of PAIR [5], finding that its design has reasonable costs, incurring no runtime overhead while imposing minimal hardware and performance overheads.

References

[1] [n. d.]. RIOT-OS. https://www.riot-os.org/

[2] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 437–448.

[3] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. 2021. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1357–1372.

[4] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. 2022. GAROTA: generalized active Root-Of-Trust architecture (for tiny embedded devices). In *31st USENIX Security Symposium (USENIX Security 22)*. 2243–2260.

[5] Anonymized. 2025. Github Repository for PAIR. To be made available after publication.

[6] ARM. 2018. Return Address Signing using ARM Pointer Authentication. https://gcc.gnu.org/legacy-ml/gcc-patches/2018-11/msg00104.html. [Online; accessed 13-February-2023].

[7] ARM. 2024. CoreSight System Trace Macrocell Technical Reference Manual. https://developer.arm.com/documentation/ddi0444/latest/

[8] Arm Ltd. 2025. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/lb. Section C6.2.49 D8.10.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*. 30–40.

[10] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[12] Adam Caulfield, Antonio Joia Neto, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2024. TRACES: TEE-based Runtime Auditing for Commodity Embedded Systems. *arXiv preprint arXiv:2409.19125* (2024).

[13] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2022. ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 721–726.

[14] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2023. ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5827–5844.

[15] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. 38–49.

[16] Alessandro Cimatti et al. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International conference on computer aided verification*. Springer, 359–364.

[17] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Vol. 2. IEEE, 119–129.

[18] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. 2022. CASU: Compromise avoidance via secure update for low-end embedded systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[19] Digilent. 2018. Basys 3 Artix-7 FPGA Trainer Board: Recommended for Introductory Users. https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/

[20] Lang Feng, Jeff Huang, Jiang Hu, and Abhijith Reddy. 2021. Fastcfi: Real-time control-flow integrity using fpga without code instrumentation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26, 5 (2021), 1–39.

[21] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. *arXiv preprint arXiv:2303.16353* (2023).

[22] Olivier Girard. 2009. openMSP430.

[23] Sashidhar Jakkamsetti, Youngil Kim, Andrew Searles, and Gene Tsudik. 2025. EILID: Execution Integrity for Low-end IoT Devices. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[24] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. PACStack: an Authenticated Call Stack.. In *USENIX Security Symposium*. 357–374.

[25] Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Patterns for secure boot and secure storage in computer systems. In *2010 International Conference on Availability, Reliability and Security*. IEEE, 569–573.

[26] Microsoft. [n. d.]. Data Execution Prevention. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[27] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2022. Survey of control-flow integrity techniques for real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 4 (2022), 1–32.

[28] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. cfi: Type-assisted control flow integrity for x86-64 binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 423–444.

[29] Muhammad Nouman Nafees, Neetesh Saxena, Alvaro Cardenas, Santiago Grijalva, and Pete Burnap. 2023. Smart grid cyber-physical situational awareness of complex operational technology attacks: A review. *Comput. Surveys* 55, 10 (2023), 1–36.

[30] Antonio Joia Neto, Adam Caulfield, and Ivan De Oliveira Nunes. 2025. RAP-Track: Efficient Control Flow Attestation via Parallel Tracking in Commodity MCUs. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[31] Antonio Joia Neto and Ivan De Oliveira Nunes. 2023. ISC-FLAT: On the Conflict Between Control Flow Attestation and Real-Time Operations. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 133–146.

[32] Antonio Joia Neto, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. 2024. PEARTS: Provable Execution in Real-Time Embedded Systems. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 47–47.

[33] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (2017), 1–33.

[34] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[35] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2020. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*. 771–788.

[36] Johannes Obermaier and Vincent Immler. 2018. 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 (2018), 289–296.

[37] James Pallister, Simon Hollis, and Jeremy Bennett. 2013. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174* (2013).

[38] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12*. Springer, 144–164.

[39] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 1–34.

[40] Gabriele Serra, Pietro Fara, Giorgiomaria Cicero, Francesco Restuccia, and Alessandro Biondi. 2022. PAC-PL: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 241–253.

[41] Xi Tan and Ziming Zhao. 2023. SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1332–1346.

[42] Tom Garrison. 2020. Intel CET Answers Call to Protect Against Common Malware Threats. https://newsroom.intel.de/editorials/intel-cet-answers-call-to-protect-against-common-malware-threats/. [Online; accessed 13-February-2023].

[43] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

[44] Moshe Y Vardi. 2005. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency: structure versus automata*. Springer, 238–266.

[45] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. 2019. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2–1.

[46] Yujie Wang, Cailani Lemieux Mack, Xi Tan, Ning Zhang, Ziming Zhao, Sanjoy Baruah, and Bryan C Ward. 2024. InsectACIDE: Debugger-based holistic asynchronous CFI for embedded system. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 360–372.

[47] Xilinx. 2017. Vivado Design Suite User Guide.

[48] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. 89–106.

[49] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[50] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd USENIX Security Symposium)*.