ASAP: Reconciling Asynchronous Real-Time Operations and Proofs of Execution in Simple Embedded Systems

Adam Caulfield
Rochester Institute of Technology

Norrathep Rattanavipanon
Prince Songkla University

Ivan De Oliveira Nunes
Rochester Institute of Technology

ABSTRACT

Embedded devices are increasingly ubiquitous and their importance is hard to overestimate. While they often support safety-critical functions (e.g., in medical devices and sensor-alarm combinations), they are usually implemented under strict cost/energy budgets, using low-end microcontroller units (MCUs) that lack sophisticated security mechanisms. Motivated by this issue, recent work developed architectures capable of generating Proofs of Execution (PoX) for the correct/expected software in potentially compromised low-end MCUs. In practice, this capability can be leveraged to provide "integrity from birth" to sensor data, by binding the sensed results/outputs to an unforgeable cryptographic proof of execution of the expected sensing process. Despite this significant progress, current PoX schemes for low-end MCUs ignore the real-time needs of many applications. In particular, security of current *PoX* schemes precludes any interrupts during the execution being proved. We argue that lack of asynchronous capabilities (i.e., interrupts within PoX) can obscure PoX usefulness, as several applications require processing real-time and asynchronous events. To bridge this gap, we propose, implement, and evaluate an **A**rchitecture for **S**ecure **A**synchronous **P**rocessing in PoX (*ASAP*). *ASAP* is secure under full software compromise, enables asynchronous *PoX*, and incurs less hardware overhead than prior work.

1 INTRODUCTION

Embedded (aka IoT or "smart") devices are increasingly popular worldwide and are becoming pervasive in all sorts of environments: from homes and offices to public spaces and industrial facilities. Not surprisingly, they are also increasingly targeted by exploits and malware. In particular, low-end micro-controller units (MCUs) are designed with strict cost, size, and energy limitations. Thus, it is hard to offer any concrete security guarantees for tasks performed by these MCUs, due to their lack of sophisticated security features, akin to those available to higher-end application processors, such as the ones used in smartphones or general-purpose controllers, e.g., Alexa or Nest. As these low-end MCUs become ubiquitous (especially in safety-critical settings), exploits that corrupt their integrity, e.g., to forge a sensed value or "lie" about having performed some expected actuation, become a significant threat.

Over the past decade, this problem was recognized and explored by the research community [24]. Previous results considered potential unauthorized software modifications/compromises in low-end devices and proposed methods to remotely verify the binary currently installed in a low-end MCU: a well known security service referred to as *Remote Attestation (R*A*)* [3, 9, 16, 22]. While $\mathcal{R}$A can prove that a remote low-end MCU is currently installed with the proper software binary, it does not provide any proofs about the correct execution of this binary (or parts thereof, i.e., functions within the binary). Therefore, recent work has focused on enhancing $\mathcal{R}$A architectures with the ability to prove the correct execution of the attested software [10], i.e., to generate Proofs of eXecution (*PoX*). The *PoX* capability, in turn, was shown to be a fundamental building block to provide additional guarantees, such as control- & data-flow attestation [11, 13]. We discuss both $\mathcal{R}$A and *PoX* in more detail in Section 2.

In addition, *PoX* can be used as a means to create sensors and actuators that "can not lie" even under the assumption that the MCU software implementing the sensing task may be compromised [10]. This is because *PoX* enables generation of unforgeable proofs for the proper execution of software tasks, including their interaction with analog peripherals via General Purpose Input/Output (GPIO) interfaces. As these proofs also bind the execution to any generated outputs (e.g., sensed values), they serve as a cryptographic proof for the integrity of the sensing process as a whole, including peripheral configuration, acquisition, and processing of the raw data.

Despite these advances, thus far *PoX* has assumed that executables must run atomically and therefore do not process interrupts. As a consequence, tasks that require handling asynchronous inputs and events (e.g., the arrival of network packets or expiring timers), cannot benefit from *PoX*. On the other hand, most real embedded applications depend on interrupts to process asynchronous events due to real-time needs. Therefore, we pose a natural question:

Are secure proofs of execution attainable for executables that must process asynchronous and real-time events/inputs?

In this paper we set out to answer this question by designing *ASAP*: an **A**rchitecture for **S**ecure **A**synchronous **P**rocessing in PoX. At a high level, the proposed design introduces two new features to an existing *PoX* architecture (APEX [10]), namely *Ephemeral Immutability and Integrity for (1) the interrupt vector table (IVT); and (2) interrupt service routines (ISRs).* These features are achieved through the selective linking of relevant ISR binaries into specific protected (and attested) memory locations; attestation of IVT; and minimal (formally verified) additional hardware support. As we discuss in the remainder of this paper, these features are sufficient to enable secure *PoX* that can also handle asynchronous events/inputs

through the use of MCU interrupts. Our evaluation shows that *ASAP* reduces the hardware overhead of existing *PoX* and incurs no additional run-time or storage/memory overhead.

2 PRELIMINARIES

2.1 Scope of Low-End MCUs

This paper focuses on tiny CPS/IoT sensors and actuators, or hybrids thereof. These are some of the smallest and weakest devices based on low-power single-core MCUs with small program and data memory (e.g., Atmel AVR ATMega, TI MSP430), with 8- and 16-bit CPUs running at 1-16MHz, with ≈ 64 KBytes of addressable memory. SRAM is used as data memory, normally ranging between 4 and 16KBytes, while the rest of the address space is available for program memory. Such devices usually run software atop "bare metal", execute instructions in place (physically from program memory), and lack memory management units (MMU) or privilege levels to support virtual memory or secure micro-kernels.

2.2 Remote Attestation ($\mathcal{R}$A)

$\mathcal{R}$A allows a trusted verifier ($\mathcal{V}$rf) to detect unauthorized binary modifications (e.g., malware infections) on an untrusted remote device, called a prover ($\mathcal{P}$rv) by remotely measuring the latter's software state. Per Fig. 1, $\mathcal{R}$A is typically realized as a challenge-response protocol with the following steps:

1)- $\mathcal{V}$rf sends an attestation request containing a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv. This request might also contain a token derived from a secret that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.

2)- $\mathcal{P}$rv receives the attestation request and computes an *authenticated integrity check* over a predefined memory region (e.g., program memory) and $\mathcal{C}$hal.

3)- $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.

4)- $\mathcal{V}$rf receives the result from $\mathcal{P}$rv, and checks whether it corresponds to a valid memory state.
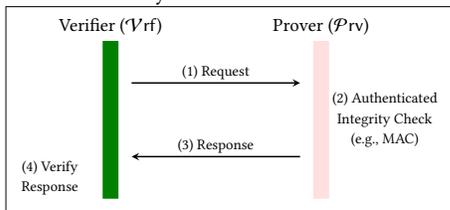
Figure 1: $\mathcal{R}$A interaction

The *authenticated integrity check* is usually realized as a Message Authentication Code (MAC) or a digital signature over $\mathcal{P}$rv memory. However, these cryptographic primitives require $\mathcal{P}$rv to have a unique secret key ($\mathcal{K}$) either shared with $\mathcal{V}$rf (MAC-s), or for which $\mathcal{V}$rf knows the public key (signatures). This $\mathcal{K}$ must reside in secure storage, and **not** be accessible to any (potentially compromised) software running on $\mathcal{P}$rv, except for trusted attestation code itself. Since most $\mathcal{R}$A threat models assume a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support.

$\mathcal{R}$A architectures fall into three categories depending on the level of hardware support: software-based, hardware-based, and hybrid. Security of software-based attestation [21, 28–30] relies on strong assumptions about precise timing and constant communication delays, which are unrealistic in the IoT/CPS ecosystem. Hardware-based methods [23, 26, 27] rely on dedicated hardware components,

e.g., TPMs [32], Intel SGX [18], or ARM TrustZone [2]. However, the cost of such hardware is prohibitive for low-end MCU-s. Hybrid $\mathcal{R}$A [3, 9, 16] aims to achieve security equivalent to hardware-based mechanisms, with low(er) hardware cost. It implements the authenticated integrity ensuring function in software, while relying on minimal hardware support to assure that this software implementation executes properly and securely.

2.3 Proofs of Execution (PoX)

PoX augments $\mathcal{R}$A's capability by proving to $\mathcal{V}$rf that: (1) the expected executable is stored in program memory, (2) this code has indeed executed, and (3) any claimed outputs were produced by its timely and authentic execution.

The first PoX architecture targeting low-end MCU-s was recently proposed in APEX [10]. APEX implements a hardware module controlling the value of a 1-bit flag called *EXEC*, which cannot be written by any software. A value *EXEC* = 1 indicates to $\mathcal{V}$rf that attested code *must* have executed successfully, between the time when the challenge $\mathcal{C}$hal was received from $\mathcal{V}$rf (recall the $\mathcal{R}$A protocol from Section 2.2) and the time when the $\mathcal{R}$A measurement occurs (via authenticated integrity ensuring function). Similarly, when it receives an attestation reply with *EXEC* = 0, $\mathcal{V}$rf can conclude that execution of said code did not occur, or that execution (or its output) was tampered with. In APEX, the $\mathcal{R}$A measurement covers: **(i)** the *EXEC* flag; **(ii)** the region where this execution's output is saved (output region – *OR*); and **(iii)** the executable itself (stored in the executable region – *ER*). Thus, security of the underlying $\mathcal{R}$A architecture guarantees that the contents of these memory regions cannot be forged/spoofed to something different from their values at the time of the attestation computation. In turn, APEX considers a code to execute properly (and sets *EXEC* = 1) if and only if:

1)- Execution is atomic and uninterrupted, from the first instruction (legal entry ER_{MIN}), to the last instruction (legal exit ER_{MAX});

2)- Neither the executable (*ER*), nor its outputs *OR* are modified in between the execution and subsequent $\mathcal{R}$A computation;

3)- During execution, data-memory (including *OR*) is not modified, by other means except for the ER execution, e.g., no modifications by other software or Direct Memory Access controllers.

From these conditions, *EXEC* = 1 assures that memory contents (of *ER* and *OR*) are consistent between *ER*'s code execution and subsequent $\mathcal{R}$A. It also assures that *ER*'s execution has integrity, e.g., it can not be corrupted by malicious interruptions that could alter its control-flow or its variables in data memory. *ER* and *OR* locations and sizes are configurable, allowing for PoX of arbitrary code and output sizes. APEX implementation builds atop the formally verified hybrid $\mathcal{R}$A architecture *VRASED* [9] and APEX hardware module is itself formally verified to adhere to a set of specifications. The conjunction of these properties are proven sufficient to imply a security notion (stated using a cryptographic security game [20]) for unforgeable proofs of execution. For brevity, we do not overview APEX proofs and refer the interested reader to [10].

As explained above, APEX mandates the absence of interrupts to guarantee that no untrusted interrupt sources and respective (potentially malicious) ISRs can interfere with the intended behavior of the executable (located in *ER*). However, this also limits the types of executable behaviors for which *PoX* is possible. In particular, it prevents provable executions from leveraging any interrupts.

In this work, we remedy this issue by enabling selected trusted ISRs, implementing intended asynchronous behavior, to be a part of provable executions without compromising *PoX* security.

3   APPLICATION EXAMPLES

Consider that $\mathcal{P}$rv is a simple MCU implementing a syringe pump that allows remote physicians to monitor and deliver medication to patients over a network, e.g., as described in [19]. Given the safety-critical and real-time nature of this application, it is paramount to verify that the execution of operations in $\mathcal{P}$rv happens as expected, e.g., that $\mathcal{P}$rv injects an *accurate and timely* dosage to the patient; otherwise, it may lead to under/overdose, affecting the patient's well-being. Such an execution can be implemented as follows:

(1)- Start injecting medication at a fixed rate;
(2)- Set up a timer interrupt according to the dosage to be injected;
(3)- Enter sleep/low-power mode;
(4)- Wake up once the timer expires and stop the injection.

To verify the correct execution of these steps, one may choose to implement $\mathcal{P}$rv using an MCU equipped with a PoX architecture such as APEX (see Section 2). However, doing so poses a challenge since this execution strictly depends on an asynchronous event (i.e., the expiring timer in step **(4)**), which in turn relies on a timer-based interrupt during execution. Unfortunately, to ensure the integrity of execution, APEX prohibits interrupts during the execution of the software being proved. Therefore, it can not be used directly to provide security/safety guarantees in this example.

To enable APEX *PoX* for this application, one simple workaround is to modify $\mathcal{P}$rv software as follows: instead of going to sleep and waking up based on the timer interrupt, the device uses the CPU to countdown, i.e., it *busy-waits* for the expected period.

Doing so eliminates the need for a timer interrupt during execution and thus allows this application to benefit from APEX *PoX*. Nonetheless, this workaround has important drawbacks. First, it imposes an unnecessary power consumption by requiring the processor to actively wait and check for the critical event. This is a significant burden for battery-powered devices (e.g., portable insulin pumps). Aside from the power consumption issue, in case of an emergency, the patient may choose to abort $\mathcal{P}$rv execution, e.g., by pressing a physical "cancel" button or by sending a network command to "abort". However, since the CPU is fully occupied and no interrupts are allowed during execution, $\mathcal{P}$rv software has no way to detect/receive and process such asynchronous safety-critical command(s). This illustrates why these approaches do not satisfy this application's safety-critical and real-time requirements.

On the other hand, simply removing the *PoX* atomicity requirement from APEX opens the door for vulnerabilities. For example, after infecting $\mathcal{P}$rv, malware may trigger an interrupt while the medication is being injected to increase the timer expiration value or, more generally, tamper with this execution by manipulating stored variables/parameters/data or its control-flow.

Aside from this example, it is not hard to find similar settings where the same real-time needs are applicable (e.g., industrial, automotive, etc). This general need motivates our work on the design of a secure *PoX* architecture that supports interrupts and thus can process asynchronous events/inputs.
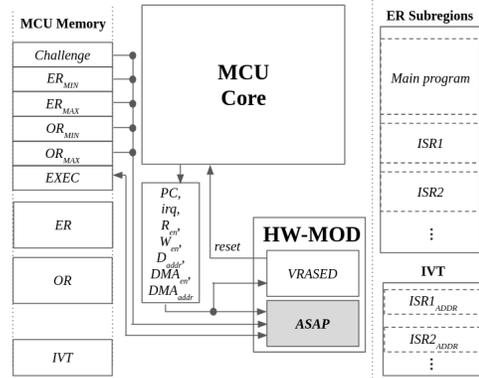

Figure 2: *ASAP* **System Architecture**

4   *ASAP* DESIGN

MCUs process asynchronous events by either (1) *busy-waiting*, i.e., actively checking in software; or (2) via hardware interrupts. As discussed in Section 3, (1) is, in many cases, not a viable approach. To prevent abuse from software external to *ER*, APEX treats any interrupt as a violation. It sets $EXEC = 0$ whenever the respective hardware signal (namely *irq*) is set, indicating an incoming interrupt during *ER*'s execution. We design *ASAP* to improve *PoX* with the ability to:

(1)- define which interrupts (and respective ISRs) are allowed and trusted as a part of *ER* behavior.
(2)- assure that the integrity of these allowed/trusted ISR(s) can be checked by $\mathcal{V}$rf as a part of the *PoX* result.
(3)- guarantee that no other untrusted/unauthorized interrupt can occur during (or tamper with) *ER* execution without $\mathcal{V}$rf detection.

Fig. 2 presents *ASAP* architecture at a high level. *ASAP* mandates that ISR binaries that are a part of *ER* behavior be placed (linked) within the *ER* memory region. As in Fig. 2, after compilation & linking, *ER* is composed of both the main program and all ISRs relevant to *ER* execution.

With that, instead of checking for the value of the *irq* to determine whether or not "some interrupt has happened", *ASAP* can check the program counter (*PC*) value. If a trusted/authorized interrupt occurs, by construction *PC* will remain inside *ER* and *ASAP* will keep $EXEC = 1$ (valid *PoX*). If an untrusted/unauthorized interrupt occurs, *PC* must leave *ER*. *ASAP* will treat the latter as a violation and set $EXEC = 0$. As the size of *ER* is configurable (by setting the values of parameters ER_{MIN} and ER_{MAX}), *ER* size can be adjusted to fit the binaries of the main program + intended ISRs.

Furthermore, when an interrupt is triggered, the MCU fetches the address of an ISR from the IVT based on the hardware trigger source (e.g, GPIO, network/UART, timer, etc). Therefore, as a part of *PoX*, it is paramount to ensure that the contents of IVT (i.e., the addresses of functions that get called due to each type of interrupt) are also attested and that the content of IVT remains consistent from the time when *ER* execution happens until when it is measured by the subsequent attestation (recall the interplay between *PoX* and $\mathcal{R}$A discussed in Section 2).

4.1   Adversary Model

We consider an adversary that controls $\mathcal{P}$rv entire software state, including code and data. It can modify any writable memory and read

any memory that is not explicitly protected by hardware-enforced access controls. Modifications to program memory can change instructions to modify the executable behavior whereas modifications to data memory can corrupt intermediate computation results or induce deviation from a program's intended control-flow. Finally, the adversary can attempt to change memory to program arbitrary interrupts before, during, or after a *PoX*.

4.2 *ASAP* Details

To enable processing of selected interrupts as a part of the *PoX*, *ASAP* modifies APEX atomicity requirements. We here go over these requirements, as well as *ASAP* modifications in detail. APEX verified properties are specified in Linear Temporal Logic (LTL), which is particularly useful for specifying and verifying sequential systems. LTL extends common logic statements with temporal quantifiers. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL includes temporal quantifiers, thus enabling sequential reasoning. In this paper, we consider the following two LTL quantifiers:

- $\mathbf{X}\phi$ – ne$\underline{\mathrm{X}}$t ϕ: holds if ϕ is true at the next system state.
- $\mathbf{G}\phi$ – $\underline{\mathrm{G}}$lobally ϕ: holds if for all future states ϕ is true.

Atomicity and Uninterruptibility of *ER* execution, as required by APEX, are formalized in LTL statements 1, 2, and 3, per [10].

$$\mathbf{G}: \{(PC \in ER) \wedge \neg(\mathbf{X}(PC) \in ER) \rightarrow PC = ER_{MAX} \vee \neg\mathbf{X}(EXEC)\} \quad (1)$$

$$\mathbf{G}: \{\neg(PC \in ER) \wedge (\mathbf{X}(PC) \in ER) \rightarrow \mathbf{X}(PC) = ER_{MIN} \vee \neg\mathbf{X}(EXEC)\} \quad (2)$$

$$\mathbf{G}: \{(PC \in ER) \wedge irq \rightarrow \neg EXEC\} \quad (3)$$

The **G** quantifier, surrounding all statements, requires them to hold at all times. LTL 1 enforces that the only way for *ER* execution to terminate without setting *EXEC* = 0 is through its last instruction: $PC = ER_{MAX}$. This is specified by checking the relation between current and next *PC* values. If the current *PC* value is within *ER* and next *PC* value is outside *ER*, then either current *PC* value is the address of ER_{MAX}, or *EXEC* is set to 0 in the next cycle. Similarly, LTL 2 uses **X** quantifier to enforce that the only way for *PC* to enter *ER* is through the very first instruction: ER_{MIN}. This prevents *ER* execution from starting at some point in the middle of *ER*, thus ensuring that *ER* always executes in its entirety. Finally, LTL 3 enforces that *EXEC* is set to zero if an interrupt happens during *ER* execution, by checking the *irq* signal. To enable selected interrupts to be triggered securely, *ASAP* removes LTL 3 and adds two new requirements [**AP1**] and [**AP2**].

[**AP1**]: *IVT Immutability & Integrity* - the memory region containing the IVT cannot be modified from the start of *ER* execution until the end of attestation. This property ensures that the attestation result correctly portrays the addresses of all ISR(s) that could have been called and processed during *ER* execution. Without this property, an adversary could modify IVT to cause an interrupt to jump to arbitrary locations within *ER* leading to violations to *ER* intended control-flow (and therefore *ER* execution integrity). This new *ASAP* property is formally specified in LTL 4, based on signals that indicate a memory write to IVT by either the CPU or DMA.

$$\mathbf{G}: \{[DMA_{en} \wedge (DMA_{addr} \in IVT)] \vee [W_{en} \wedge (D_{addr} \in IVT)] \rightarrow \neg EXEC\} \quad (4)$$
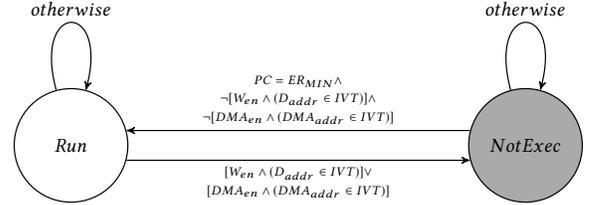

Figure 3: Verified FSM for LTL 4 for IVT Immutability.

In LTL 4, W_{en} is a CPU signal that indicates that a CPU memory write is happening to the address in the D_{addr} signal. DMA_{en} and DMA_{addr} serve the same purpose for detecting writes by DMA to specific locations. *EXEC* is set to 0 whenever there is a CPU write or DMA access to IVT.

Fig. 3 depicts a Verilog FSM implemented and verified to comply with LTL 4 ([**AP1**]). The FSM has two states: *Run* and *NotExec*. The FSM transitions to the *NotExec* state and outputs *EXEC* = 0 whenever a violation happens, i.e., whenever *IVT* is modified. It transitions back to *Run* when *ER*'s execution is restarted ($PC = ER_{MIN}$).

[**AP2**]: *ISR Immutability* - trusted/authorized ISR binaries cannot be modified between the start of *ER* execution until attestation is completed. This is required to ensure that the attestation result correctly reflects the behavior implemented by the ISRs as a part of *ER*. Without this property, the adversary could overwrite an authorized ISR arbitrarily, modifying its behavior without detection by $\mathcal{V}$rf. Since APEX already enforces *ER* immutability between execution and attestation, *ASAP* reuses this support by simply placing (linking) the trusted ISR binaries to be within *ER*.

***ASAP* Security Argument:** *Let ER contain a program composed of a main task and its trusted ISR(s) which can be asynchronously executed due to their respective interrupt triggers. Once execution starts ($PC = ER_{MIN}$), APEX ensures that the PoX result will reflect EXEC = 1 iff: (1) ER is not modified until both its execution and subsequent attestation are over; (2) no external execution: PC stays within ER until it reaches ER_{MAX} (LTL 1); and (3) OR is not modified in between execution and attestation completion. Per [**AP1**], IVT is also immutable after ER execution starts (otherwise ASAP sets EXEC = 0) and is attested. Hence, the PoX result includes a report detailing which code section is executed due to each interrupt source in the system (i.e., the IVT configuration). Finally, due to [**AP2**], ISRs relevant to ER execution are all contained within ER, making them immutable and attested. Therefore, the PoX result allows $\mathcal{V}$rf to check that all IVT entries that point to an address within ER correspond to the entry point of an intended/expected ISR binary. Additionally, any execution of an unauthorized/untrusted ISR requires jumping outside ER, which sets EXEC = 0 (per LTL 1), resulting in an invalid PoX.*

5 IMPLEMENTATION & EVALUATION

We implemented *ASAP* on OpenMSP430: an open-source design for the MSP430 architecture, which represents the targeted class of devices discussed in Section 2.1. *ASAP* builds on top of APEX, which in turn relies on VRASED $\mathcal{R}A$ architecture. As shown in Fig. 2, *ASAP* is implemented within the module labeled HW-Mod. Its features are attained by small hardware modifications. *ASAP* is publicly available at [**?**].

(a) Linker Example   (b) Software Example

Figure 4: Software and Linking to achieve ISR immutability

To achieve **[AP1] IVT Immutability**, the hardware is extended with the module shown in Fig. 3, used to detect any writes to IVT. In OpenMSP430, IVT is stored in a fixed physical location, i.e., in the last 32-byte of addressable memory: from the base address 0xFFE0 to last address 0xFFFF.

To selectively link trusted ISRs, as required by **[AP2] ISR Immutability**, we implement *ER* linking as in Fig. 4. In it, a sample "dummy function" executes a loop, and one ISR is implemented to write to GPIO PORT5 when any asynchronous signal is received from GPIO PORT1 (e.g., a button press). The header of these functions assigns them with the section label "exec.body". This label in conjunction with a modified linker script for MSP430 allows for these functions to be placed inside *ER* region. Labels "exec.start" and "exec.leave" are used to determine *ER* entry point (ER_{MIN}) and exit point (ER_{MAX}). In this example, these functions are named *startER()* and *exitER()*. *startER()* simply calls "dummy function" (i.e., the program's behavior) and *exitER()* simply returns, i.e., concludes the provable execution. These functions have section labels "exec.start" and "exec.leave" so that they can be identified and placed at the beginning and end of *ER* by the linker script shown in Fig. 4(a).

With this design in place, experiments were conducted to demonstrate the differences between APEX and *ASAP* when processing interrupts during *ER* execution. Fig. 5 shows simulation wave-forms for three cases including APEX and *ASAP*. In each figure, the following signals are depicted over time: ER_{MIN}, ER_{MAX}, $EXEC$, irq (interrupt request signal), and PC.

Fig. 5(a) shows the behavior of *ASAP* in the instance that a legitimate and authorized interrupt occurs (i.e., its corresponding ISR lies within ER) while executing ER. As shown, *ER* is currently executing, as the PC value is between ER_{MIN} and ER_{MAX}. As the signal irq is set, indicating an interrupt, PC jumps from the main program at $0xE1AC$ to the ISR first instruction at $0xE1B0$. Since the destination is still within the range of the *ER*, the $EXEC$ signal is unaffected and remains 1. As a result, a subsequent attestation would convey to $\mathcal{V}$rf that execution was successful and untampered with. Fig. 5(b) shows the behavior of *ASAP* under the influence of an external interrupt that has not been authorized to be a part of *ER* behavior and therefore not linked within *ER*. In this scenario, initially PC is also within *ER*. However, once the external interrupt is handled, PC jumps to the ISR located outside of *ER* (at $0xE0D6$). In accordance with LTLs 1 and 2, $EXEC$ is set to 0. In APEX, shown in Fig. 5(c), any irq causes $EXEC = 0$, regardless of the PC value or whether or not the ISR is located within *ER* and a part of the executable behavior. This illustrates *ASAP* ability to separate trusted and untrusted

interrupts and handle trusted interrupts while keeping *PoX* secure against untrusted ones.

To demonstrate *ASAP* practicality, we synthesized and implemented its RTL design on an Artix-7 FPGA (Basys3 prototyping board). We note that a hardware design that is synthesizable on FPGA can also be used to manufacture an Application-Specific-Integrated-Circuit (ASIC) for large-scale usage. Below we report on *ASAP* costs based on this prototype.

Hardware and Memory Overhead. To evaluate *ASAP* hardware overhead, we compare it to APEX in Fig. 6. Similar to related work [9, 10, 12, 14, 15], we consider the hardware overhead in terms of additional Look-up Tables (LUTs) and registers. The increase in LUTs is an estimate of the additional chip cost and size required for combinatorial logic, while the number of extra registers indicates additional states required by the sequential logic in *ASAP* FSMs. Fig. 6 shows that *ASAP* utilizes 24 less LUTs and 3 less registers than APEX. As **[AP2]** reuses existent *ER* protection to ensure immutability of ISRs, it incurs no additional hardware overhead. Additionally, APEX requires monitoring the *irq* signal, which is propagated into several sub-modules to enforce LTL 3. Because this is no longer required in *ASAP*, there is a reduction in the register and LUT utilization, despite the need for an additional 2-state FSM to enforce **[AP1]**.

Runtime Overhead. Neither *ASAP* nor APEX incur additional execution time for the tasks being proved, as no instrumentation or additional instructions are required. This is because relevant runtime security properties and control of the *EXEC* flag are implemented by hardware that runs in parallel with the CPU (as in Fig. 2). Linking in **[AP2]** is static and done at compilation time.

Verification Cost. We verified *ASAP* on a 64-bit Ubuntu 18.04 machine with an Intel i7 3.6GHz CPU using NuSMV [7] model-checker to show it adheres to the new property and still maintains all other guarantees required by APEX. *ASAP* verification takes ≈150s for a total of 21 LTL properties and requires 96MB of RAM. *ASAP* verified implementation totals 2155 lines in the Verilog HDL.

6 RELATED WORK

$\mathcal{R}$A **& Interrupts.** One important security property of $\mathcal{R}$A (see Section 2) is *temporal consistency*, i.e., guaranteeing that an $\mathcal{R}$A result always reflects an instantaneous snapshot of $\mathcal{P}$rv attested memory. Lack thereof allows malware to escape detection by copying and/or erasing itself during $\mathcal{R}$A. Temporal consistency is usually achieved by enforcing atomic (uninterruptible) $\mathcal{R}$A execution. However, since $\mathcal{R}$A is often used in safety-critical and/or real-time settings [5], the atomicity requirement might interfere with the MCU applications. To address this issue, SMARM [6] allows $\mathcal{R}$A to be interruptible by using probabilistic malware detection. Meanwhile, ERASMUS [4] and SeED [17] are based on periodic self-measurements in order to detect transient malware that infects $\mathcal{P}$rv and leaves before the next $\mathcal{R}$A instance. RATA [12] actively monitors writes to program memory to detect such attacks. We note that these efforts should not be confused with interruptable *PoX*. $\mathcal{R}$A by itself does not provide any runtime guarantees (see Section 2) but rather serves as a building block for more expressive proofs such as *PoX*.

Proof of Execution (PoX). Prior work has focused on providing PoX in high-end devices. Flicker [25] leverages TPM and hardware extensions to implement a PoX functionality on Intel
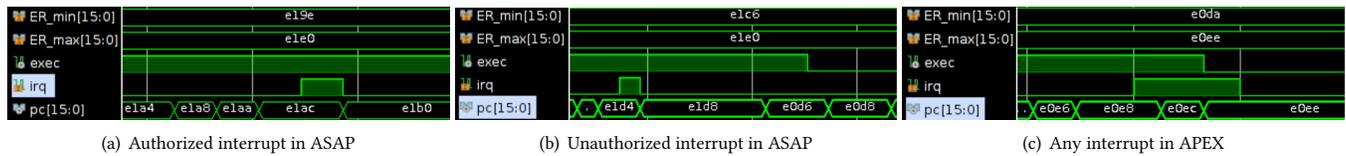
(a) Authorized interrupt in ASAP

(b) Unauthorized interrupt in ASAP

(c) Any interrupt in APEX

Figure 5: Comparison: interrupt handling in *ASAP* vs. APEX

(a) Total extra Look-Up Tables (LUTs)
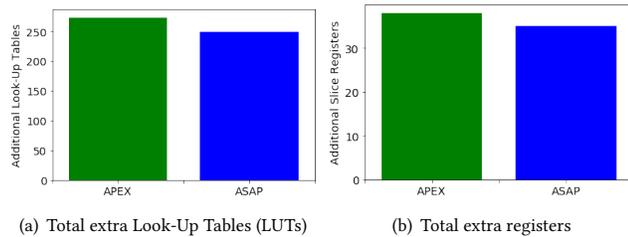
(b) Total extra registers

Figure 6: Overhead comparison between APEX and *ASAP*

and AMD computers. Sanctum [8] implements PoX in Intel-SGX-like devices by instrumenting enclaved code to output information about its own execution to a remote $\mathcal{V}$rf. Both of these approaches rely on complex hardware that is unavailable in low-end embedded systems. Thus far, APEX [10] is the only PoX architecture designed for simple MCUs. However, in order to successfully produce an unforgeable proof, APEX requires execution to run without interruptions, precluding its usefulness on interrupt-based applications.

Control-Flow & Data-Flow Attestation (CFA/DFA). The goal of CFA [1, 14] is to measure the exact control flow path taken during execution on $\mathcal{P}$rv. The result of this measurement can convey to $\mathcal{V}$rf the order in which instructions of some specific binary have executed on $\mathcal{P}$rv. DFA [15, 31] augments CFA with the ability to detect data-only attacks. Recent work [11, 13], built on top of APEX to obtain CFA and DFA in low-end MCUs. As such, they are also limited to APEX's uninterruptability requirement and can not process asynchronous inputs during execution/CFA/DFA. We believe that our work addresses this limitation.

7 CONCLUSION

This work is motivated by much needed integrity assurance for execution in safety-critical edge devices, which are often implemented with low-power and low-cost MCUs. Existing mechanisms for producing unforgeable proofs of execution in low-end MCUs require executables not to handle interrupts, precluding their uses in many real-time/mission-critical settings. To address this issue, we proposed *ASAP*: the first architecture able to generate *PoX* for software that implements interrupts within its behavior. *ASAP* extends the original APEX *PoX* architecture to securely convey information about the ephemeral immutability and integrity of the IVT and relevant ISRs. We show that these two properties are sufficient to realize *ASAP* securely and that they can be obtained through the appropriate linking of ISRs into protected memory and minimal additional verified hardware support, on top of that already provided by APEX. Our experimental results show *ASAP* feasibility and affordability, even on some of the lowest-end MCUs.

REFERENCES
[1] T. Abera et al., "C-flat: Control-flow attestation for embedded systems software," in *ACM CCS*, 2016.
[2] Arm Ltd., "Arm TrustZone," 2018. [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone
[3] F. Brasser et al., "Tytan: Tiny trust anchor for tiny devices," in *DAC*.   ACM, 2015.
[4] X. Carpent et al., "ERASMUS: Efficient remote attestation via self-measurement for unattended settings," in *DATE*, 2018.
[5] ——, "Reconciling remote attestation and safety-critical operation on simple iot devices," in *DAC*, 2018.
[6] ——, "Remote attestation of iot devices via SMARM: Shuffled measurements against roving malware," in *IEEE HOST*, 2018.
[7] A. Cimatti et al., "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, 2002.
[8] V. Costan et al., "Sanctum: Minimal hardware extensions for strong software isolation," in {*USENIX*} *Security*, 2016.
[9] I. De Oliveira Nunes et al., "VRASED: A verified hardware/software co-design for remote attestation," *USENIX Security*, 2019.
[10] ——, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *USENIX Security*, 2020.
[11] ——, "Dialed: Data integrity attestation for low-end embedded devices," *DAC*, 2021.
[12] ——, "On the toctou problem in remote attestation," in *ACM CCS*, 2021.
[13] ——, "Tiny-cfa: A minimalistic approach for control flow attestation using verified proofs of execution." in *DATE*, 2021.
[14] G. Dessouky et al., "Lo-fat: Low-overhead control flow attestation in hardware," in *DAC*, 2017.
[15] ——, "Litehax: Lightweight hardware-assisted attestation of program execution," in *ICCAD*, 2018.
[16] K. Eldefrawy et al., "Smart: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.
[17] A. Ibrahim et al., "SeED: secure non-interactive attestation for embedded devices," in *ACM WiSec*, 2017.
[18] Intel, "Intel Software Guard Extensions (Intel SGX)." [Online]. Available: https://software.intel.com/en-us/sgx
[19] M. R. Islam et al., "Design and implementation of low cost smart syringe pump for telemedicine and healthcare," in *ICREST*, 2019.
[20] J. Katz and Y. Lindell, *Introduction to modern cryptography*.   CRC press, 2014.
[21] R. Kennell et al., "Establishing the genuinity of remote computer systems," in *USENIX Security*, 2003.
[22] P. Koeberl et al., "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.
[23] X. Kovah et al., "New results for timing-based attestation," in *IEEE S&P '12*, 2012.
[24] B. Kuang et al., "A survey of remote attestation in internet of things: Attacks, countermeasures, and prospects," *Computers & Security*, 2022.
[25] J. McCune et al., "Flicker: An execution infrastructure for tcb minimization," in *EuroSys*, 2008.
[26] J. Petroni et al., "Copilot — A coprocessor-based kernel runtime integrity monitor," in *USENIX Security*, 2004.
[27] D. Schellekens et al., "Remote attestation on legacy operating systems with trusted platform modules," *Science of Comp. Programming*, 2008.
[28] A. Seshadri et al., "SWATT: Software-based attestation for embedded devices," in *IEEE S&P '04*, 2004.
[29] ——, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SOSP*, 2005.
[30] ——, "SAKE: software attestation for key establishment in sensor networks," in *DCOSS*, 2008.
[31] Z. Sun et al., "Oat: Attesting operation integrity of embedded devices," in *IEEE S&P*, 2020.
[32] TCG, "Trusted platform module (tpm)," 2017. [Online]. Available: http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/