APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise

Ivan De Oliveira Nunes\textsuperscript{1}, Karim Eldefrawy\textsuperscript{2}, Norrathep Rattanavipanon\textsuperscript{1,3}, and Gene Tsudik\textsuperscript{1}

\textsuperscript{1}University of California, Irvine
\textsuperscript{2}SRI International
\textsuperscript{3}Prince of Songkla University, Phuket Campus
ivanoliv@uci.edu, karim.eldefrawy@sri.com, norrathep.r@phuket.psu.ac.th, gene.tsudik@uci.edu

Abstract
Modern society is increasingly surrounded by, and is growing accustomed to, a wide range of Cyber-Physical Systems (CPS), Internet-of-Things (IoT), and smart devices. They often perform safety-critical functions, e.g., personal medical devices, automotive CPS as well as industrial and residential automation, e.g., sensor-alarm combinations. On the lower end of the scale, these devices are small, cheap and specialized sensors and/or actuators. They tend to host small anemic CPUs, have small amounts of memory and run simple software. If such devices are left unprotected, consequences of forged sensor readings or ignored actuation commands can be catastrophic, particularly, in safety-critical settings. This prompts the following three questions: (1) How to trust data produced, or verify that commands were performed, by a simple remote embedded device?, (2) How to bind these actions/results to the execution of expected software? and, (3) Can (1) and (2) be attained even if all software on a device can be modified and/or compromised?

In this paper we answer these questions by designing, demonstrating security of, and formally verifying, APEX: an Architecture for Provable Execution. To the best of our knowledge, this is the first of its kind result for low-end embedded systems. Our work has a range of applications, especially, automotive systems and public venues. Tasks performed by these devices are often safety-critical. For example, temperature, pressure, humidity, speed) reported by sensors, and on actions taken by actuators, such as: turning on the A/C, sounding an alarm, or reducing speed.

A cyber-physical control system is usually composed of multiple sensors and actuators, at the core of each is a low-cost micro-controller unit (MCU). Such devices typically run simple software, often on "bare metal", i.e., with no microkernel or hypervisor. They tend to be operated by a remote central control unit and despite their potential importance to overall system functionality, low-end devices are typically designed to minimize cost, physical size and energy consumption, e.g., TI MSP430.

Therefore, their architectural security is usually primitive or non-existent, thus making them vulnerable to malware infections and other malicious software modifications. A compromised MCU can spoof sensed quantities or ignore actuation commands, leading to potentially catastrophic results. For example, in a smart city, large-scale erroneous reports of electricity consumption by smart meters might lead to power outages. A medical device that returns incorrect values when queried by a remote physician might result in a wrong drug being prescribed to a patient. A compromised car engine temperature sensor that reports incorrect (low) readings can lead to undetected overheating and major damage. However, despite very real risks of remote software compromise, most users believe that these devices execute expected software and thus perform their expected function.

In this paper, we argue that Proofs of Execution (PoX) are both important and necessary for securing low-end MCUs. Specifically, we demonstrate in Section 7.3, that PoX schemes can be used to construct sensors and actuators that “can not lie”, even under the assumption of full software compromise. In a nutshell, a PoX conveys that an untrusted remote (and possibly compromised) device really executed specific software, and all execution results are authenticated and cryptographically bound to this execution. This functionality is similar to authenticated outputs that can be produced by software execution in SGX-alike architectures \cite{13,25} on high-end devices, such as desktops and servers.

One key building block in designing PoX schemes is Remote Attestation (RA). Basically, RA is a means to detect malware on a remote low-end MCU. It allows a trusted verifier (Vrf) to remotely measure memory contents (or software state) of an
untrusted embedded device (Prv). RA is usually realized as a
2-message challenge-response protocol:
1. Vrf sends an attestation request containing a challenge
   (Chal) to Prv. It might also contain a token derived from a
   secret (shared by Vrf and Prv) that allows Prv to au-
   thenticate Vrf.
2. Prv receives the attestation request, authenticates the to-
   ken (if present) and computes an authenticated integrity
   check over its memory and Chal. The memory region can
   be either pre-defined, or explicitly specified in the request.
3. Prv returns the result to Vrf.
4. Vrf receives the result, and decides whether it corre-
   sponds to a valid memory state.

The authenticated integrity check is typically implemented as
a Message Authentication Code (MAC) computed over Prv
memory. We discuss one concrete RA architecture in Section 3.

Despite major progress and many proposed RA architectures
with different assumptions and guarantees [6–8, 15, 19, 20, 29,
33, 35, 36, 39], RA alone is insufficient to obtain proofs of
execution. RA allows Vrf to check integrity of software re-
 siding in the attested memory region on Prv. However, by
itself, RA offers no guarantee that the attested software is ever
executed or that any such execution completes successfully.
Even if the attested software is executed, there is no guarantee
that it has not been modified (e.g., by malware residing else-
where in memory) during the time between its execution and
its attestation. This phenomenon is well known as the Time-Of-
Check-Time-Of-Use (TOCTOU) problem. Finally, RA does
not guarantee authenticity and integrity of any output produced
by the execution of the attested software.

To bridge this gap, we design and implement APEX: an
Architecture for Provable Execution. In addition to RA, APEX
allows Vrf to request an unforgeable proof that the attested
software executed successfully and (optionally) produced cer-
tain authenticated output. These guarantees hold even in case
of full software compromise on Prv. Contributions of this work
include:
– New security service: we design and implement APEX for
  unforgeable remote proofs of execution (PoX). APEX is com-
  posed with VRASED [15], a formally verified hybrid RA ar-
  chitecture. As discussed in the rest of this paper, obtaining
  provably secure PoX requires significant architectural support
  on top of a secure RA functionality (see Section 7). Nonethe-
  less, we show that, by careful design, APEX achieves all neces-
  sary properties of secure PoX with fairly low overhead. To the
  best of our knowledge, this is the first security architecture for
  proofs of remote software execution on low-end devices.
– Provable security & implementation verification: secure
  PoX involves considering, and reasoning about, several details
  which can be easily overlooked. Ensuring that all necessary
  PoX components are correctly implemented, composed, and
  integrated with the underlying RA functionality is not trivial. In
  particular, early RA architectures oversimplified PoX require-
  ments, leading to the incorrect conclusion that PoX can be
  obtained directly from RA; see examples in Section 2. In this
  work, we show that APEX yields a secure PoX architecture.
  All security properties expected from APEX implementation
  are formally specified using Linear Temporal Logic (LTL) and
  APEX modules are verified to adhere to these properties. We
  also prove that the composition of APEX new modules with a
  formally verified RA architecture (VRASED) implies a concrete
definition of PoX security.
– Evaluation, publicly available implementation and applica-
tions: APEX was implemented on a real-world low-end
MCU (TI MSP430) and deployed using commodity FPGAs.
Both design and verification are publicly available at [1]. Our
evaluation shows low hardware overhead, affordable even
for low-end MCUs. The implementation is accompanied by
a sample PoX application; see Section 7.3. As a proof of
concept, we use APEX to construct a trustworthy safety-critical
device, whereupon malware can not spoof execution results
(e.g., fake sensed values) without detection.

Targeted Devices & Scope: This work focuses on CPS/IoT
sensors and actuators with relatively weak computing power.
They are some of the lowest-end devices based on low-power
single core MCUs with only a few KBytes of program and
data memory. Two prominent examples are: TI MSP430 and
Atmel AVR ATmega. These are 8- and 16-bit CPUs, typically
running at 1-16MHz clock frequencies, with ≈ 64 KBytes of
addressable memory. SRAM is used as data memory and its
size is normally ranges from 4 to 16KBytes, with the rest of
address space available for program memory. These devices
execute instructions in place (in physical memory) and have no
memory management unit (MMU) to support virtual memory.
Our implementation focuses on MSP430. This choice is due to
public availability of a well-maintained open-source MSP430
hardware design from Open Cores [23]. Nevertheless, our
machine model and the entire methodology developed in this
paper are applicable to other low-end MCUs in the same class,
such as Atmel AVR ATmega.

2 Related Work
Remote Attestation (RA)– architectures fall into three cate-
grories: hardware-based, software-based, or hybrid. Hardware-
based [31, 37, 42] relies on dedicated secure hardware compo-
nents, e.g., Trusted Platform Modules (TPMs) [42]. However,
the cost of such hardware is normally prohibitive for low-end
IoT/CPS devices. Software-based attestation [27, 40, 41] re-
quires no hardware security features but imposes strong secu-
rity assumptions about communication between Prv and Vrf,
which are unrealistic in the IoT/CPS ecosystem (though, it is
the only choice for legacy devices). Hybrid RA [7, 19, 21, 22, 30]
aims to achieve security equivalent to hardware-based mecha-
nisms at minimal cost. It thus entails minimal hardware require-
ments while relying on software to reduce overall complexity
and RA footprint on Prv.
The first hybrid RA architecture – SMART [20] – acknowledged the importance of proving remote code execution on \( P_{rv} \), in addition to just attesting \( P_{rv} \)’s memory. Using an attest-then-execute approach (see Algorithm 4 in [20]), SMART attempts to provide software execution by specifying the address of the first instruction to be executed after completion of attestation. However, SMART offers no guarantees beyond “invoking the executable”. It does not guarantee that execution completes successfully or that any produced outputs are tied to this execution. For example, SMART can not detect if execution is interrupted (e.g., by malware) and never resumed. A reset (e.g., due to software bugs, or \( P_{rv} \) running low on power) might happen after invoking the executable, preventing its successful completion. Also, direct memory access (DMA) can occur during execution and it can modify the code being executed, its intermediate values in data memory, or its output. SMART neither detects nor prevents DMA-based attacks, since it assumes DMA-disabled devices.

Another notable RA architecture is TrustLite [29], which builds upon SMART to allow secure interrupts. TrustLite does not enforce temporal consistency of attested memory; it is thus conceptually vulnerable to self-relocating malware and memory modification during attestation [9]. Consequently, it is challenging to deriving secure PoX from TrustLite. Several other prominent low-to-medium-end RA architectures – e.g., SANCUS [35], HYDRA [19], and TyTaN [7] – do not offer PoX. In this paper, we show that the execute-then-attest approach, using a temporally consistent RA architecture, can be designed to provide unforgeable proofs of execution that are only produced if the expected software executes correctly and its results are untampered.

Control Flow Attestation (CFA)– In contrast with RA, which measures \( P_{rv} \)’s software integrity, CFA techniques [2, 16, 17, 44] provide \( P_{rf} \) with a measurement of the exact control flow path taken during execution of specific software on \( P_{rv} \). Such measurements allow \( P_{rf} \) to detect run-time attacks. We believe that it is possible to construct a PoX scheme that relies on CFA to produce proofs of execution based on the attested control flow path. However, in this paper, we advocate a different approach – specific for proofs of execution – for two main reasons:

- CFA requires substantial additional hardware features in order to attest, in real time, executed instructions along with memory addresses and the program counter. For example, C-FLAT [2] assumes ARM TrustZone, while LO-FAT [17] and LiteHAX [16] require a branch monitor and a hash engine. We believe that such hardware components are not viable for low-end devices, since their cost (in terms of price, size, and energy consumption) is typically higher than the cost of a low-end MCU itself. For example, the cheapest Trusted Platform Module (TPM) [42], is about 10\( \times \) more expensive than MSP430 MCU itself\(^1\).

As shown in Section 7.2, current CFA architectures are also considerably more expensive than the MCU itself and hence not realistic in our device context.

- CFA assumes that \( P_{rf} \) can enumerate a large (potentially exponential!) number of valid control flow paths for a given program, and verify a valid response for each. This burden is unnecessary for determining if a proof of execution is valid, because one does not need to know the exact execution path in order to determine if execution occurred (and terminated) successfully; see Section 4.1 for a discussion on run-time threats.

Instead of relying on CFA, our work constructs a PoX-specific architecture – APEX – that enables low-cost PoX for low-end devices. APEX is non-invasive (i.e., it does not modify MCU behavior and semantics) and incurs low hardware overhead: around 2\% for registers and 12\% for LUTs. Also, \( P_{rf} \) is not required to enumerate valid control flow graphs and the verification burden for PoX is exactly the same as the effort to verify a typical remote attestation response for the same code.

Formally Verified Security Services– In recent years, several efforts focused on formally verifying security-critical systems. In terms of cryptographic primitives, Hawblitzel et al. [24] verified implementations of SHA, HMAC, and RSA. Bond et al. [5] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. Zinzindohoué, et al. [45] developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [3]. Larger security-critical systems have also been successfully verified. Bhargavan [4] implemented the TLS protocol with verified cryptographic security. CompCert [32] is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. [28] designed and proved functional correctness of the seL4 microkernel. More recently, VRased [15] realized a formally verified hybrid RA architecture. APEX architecture, proposed in this paper, uses VRased RA functionality (see Section 3.2 for details) composed with additional formally verified architectural components to obtain provably secure PoX.

Proofs of Execution (PoX)– Flicker [34] offers a means for obtaining PoX in high-end devices. It uses TPM-based attestation and sealed storage, along with late launch support offered by AMD’s Secure Virtual Machine extensions [43] to implement an infrastructure for isolated code execution and attestation of the executed code, associated inputs, and outputs. Sanctum [13] employs a similar approach by instrumenting Intel SGX’s enclave code to convey information about its own execution to a remote party. Both of these approaches are only suitable for high-end devices and not for low-end devices targeted in this paper. As discussed earlier, no prior hybrid RA architecture for low-end devices provides PoX.

\(^1\) Source: https://www.digikey.com/
3 Background

3.1 Formal Verification, Model Checking & Linear Temporal Logic

Computer-aided formal verification typically involves three basic steps. First, the system of interest (e.g., hardware, software, communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system retains them. This can be achieved by either Theorem Proving or Model Checking. In this work, we use the latter to verify the implementation of system modules, and the former to derive new properties from sub-properties that were proved for the modules’ implementation.

In one instantiation of model checking, properties are specified as formulae using Temporal Logic (TL) and system models are represented as FSMs. Hence, a system is represented by a triple \((S, S_0, T)\), where \(S\) is a finite set of states, \(S_0 \subseteq S\) is the set of possible initial states, and \(T \subseteq S \times S\) is the transition relation set – it describes the set of states that can be reached in a single step from each state. The use of TL to specify properties allows representation of expected system behavior over time.

We apply the widely used model checker NuSMV [11], which can be used to verify generic HW or SW models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV model specification is simple. Furthermore, it can be automated [26], because the standard RTL design already relies on describing hardware as an FSM.

In NuSMV, properties are specified in Linear Temporal Logic (LTL), which is particularly useful for verifying sequential systems, since LTL extends common logic statements with temporal clauses. In addition to propositional connectives, such as conjunction (\(\land\)), disjunction (\(\lor\)), negation (\(\neg\)), and implication (\(\rightarrow\)), LTL includes temporal connectives, thus enabling sequential reasoning. In this paper, we are interested in the following temporal connectives:

- \(X\phi\) – next \(\phi\): holds if \(\phi\) is true at the next system state.
- \(F\phi\) – Future \(\phi\): holds if there exists a future state where \(\phi\) is true.
- \(G\phi\) – Globally \(\phi\): holds if for all future states \(\phi\) is true.
- \(\phi U \psi\) – \(\phi\) Until \(\psi\): holds if there is a future state where \(\psi\) holds and \(\phi\) holds for all states prior to that.
- \(\phi B \psi\) – \(\phi\) Before \(\psi\): holds if the existence of state where \(\psi\) holds implies the existence of an earlier state where \(\phi\) holds. This connective can be expressed using \(U\) through the equivalence: \(\phi B \psi \equiv \neg(\neg\phi U \psi)\).

This set of temporal connectives combined with propositional connectives (with their usual meanings) allows us to specify powerful rules. NuSMV works by checking LTL specifications against the system FSM for all reachable states in such FSM.

3.2 Formally Verified RA

VRASED [15] is a formally verified hybrid (hardware/software co-design) RA architecture, built as a set of sub-modules, each guaranteeing a specific set of sub-properties. All VRASED sub-modules, both hardware and software, are individually verified. Finally, the composition of all sub-modules is proved to satisfy formal definitions of RA soundness and security. RA soundness guarantees that an integrity-ensuring function (HMAC in VRASED’s case) is correctly computed on the exact memory being attested. Moreover, it guarantees that attested memory remains unmodified after the start of RA computation, protecting against “hide-and-seek” attacks caused by self-relocating malware [9]. RA security ensures that RA execution generates an unforgeable authenticated memory measurement and that the secret key \(K\) used in computing this measurement is not leaked before, during, or after, attestation.

To achieve aforementioned goals, VRASED software (SW-Att) is stored in Read-Only Memory (ROM) and relies on a formally verified HMAC implementation from HACL* cryptographic library [45]. A typical execution of SW-Att is carried out as follows:

1. Read challenge \(Chal\) from memory region \(MR\).
2. Derive a one-time key from \(Chal\) and the attestation master key \(K\) using an HMAC-based Key Derivation Function (KDF).
3. Generate an attestation token \(H\) by computing an HMAC over an attested memory region \(AR\) using the derived key:
   \[ H = HMAC(KDF(K, MR), AR) \]
4. Write \(H\) into \(MR\) and return the execution to unprivileged software, i.e, normal applications.

VRASED hardware (HW-Mod) monitors 7 MCU signals:

- \(PC\): Current Program Counter value;
- \(Ren\): Signal that indicates if the MCU is reading from memory (1-bit);
- \(Wen\): Signal that indicates if the MCU is writing to memory (1-bit);
- \(D_{addr}\): Address for an MCU memory access;
- \(DMA_{en}\): Signal that indicates if Direct Memory Access (DMA) is currently enabled (1-bit);
- \(DMA_{addr}\): Memory address being accessed by DMA.
- \(irq\): Signal that indicates if an interrupt is happening (1-bit);

These signals are used to determine a one-bit \(reset\) signal output. Whenever \(reset\) is set to 1 a system-wide MCU reset is triggered immediately, i.e., before the execution of the next instruction. This condition is triggered whenever VRASED’s hardware detects any violation of its security properties. VRASED hardware is described in Register Transfer Level (RTL) using Finite State Machines (FSMs). Then, NuSMV Model Checker [12] is used to automatically prove that such FSMs achieve claimed security sub-properties. Finally, the proof that the conjunction of hardware and software sub-properties implies end-to-end soundness and security is done using an LTL theorem prover.
More formally, VRASED end-to-end security proof guarantees that no probabilistic polynomial time (PPT) adversary can win the RA security game (See Definition 7 in Appendix B) with non-negligible probability in terms of the security parameter.

4 Proof of Execution (PoX) Schemes

A Proof of Execution (PoX) is a scheme involving two parties: (1) a trusted verifier \( \mathcal{H} \), and (2) an untrusted (potentially infected) remote prover \( \mathcal{P} \). Informally, the goal of PoX is to allow \( \mathcal{H} \) to request execution of specific software \( S \) by \( \mathcal{P} \). As part of PoX, \( \mathcal{P} \) must reply to \( \mathcal{H} \) with an authenticated unforgeable cryptographic proof (H) that convinces \( \mathcal{H} \) that \( \mathcal{P} \) indeed executed \( S \). To accomplish this, verifying \( \mathcal{H} \) must prove that: (1) \( S \) executed atomically, in its entirety, and that such execution occurred on \( \mathcal{P} \) (and not on some other device); and (2) any claimed result/output value of such execution, that is accepted as legitimate by \( \mathcal{H} \), could not have been spoofed or modified. Also, the size and behavior (i.e., instructions) of \( S \), as well as the size of its output (if any), should be configurable and optionally specified by \( \mathcal{H} \). In other words, PoX should provide proofs of execution for arbitrary software, along with corresponding authenticated outputs. Definition 1 specifies PoX schemes in detail.

We now justify the need to include atomic execution of \( S \) in the definition of PoX. On low-end MCUs, software typically runs on “bare metal” and, in most cases, there is no mechanism to enforce memory isolation between applications. Therefore, allowing \( S \) execution to be interrupted would permit other (potentially malicious) software running on \( \mathcal{P} \) to alter the behavior of \( S \). This might be done, for example, by an application that interrupts execution of \( S \) and changes intermediate computation results in \( S \) data memory, thus tampering with its output or control flow. Another example is an interrupt that resumes \( S \) at different instruction modifying \( S \) execution flow. Such actions could modify \( S \) behavior completely via return oriented programming (ROP).

4.1 PoX Adversarial Model & Security Definition

We consider an adversary \( \mathcal{A} \) that controls \( \mathcal{P} \)'s entire software state, code, and data. \( \mathcal{A} \) can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules. \( \mathcal{A} \) may also have full control over all Direct Memory Access (DMA) controllers of \( \mathcal{P} \). Recall that DMA allows a hardware controller to directly access main memory (e.g., RAM, flash or ROM) without going through the CPU.

We consider a scheme PoX = (XRequest, XAtomicExec, XPProve, XVerify) to be secure if the aforementioned \( \mathcal{A} \) has only negligible probability of convincing \( \mathcal{H} \) that \( S \) executed successfully when, in reality, such execution did not take place, or was interrupted. In addition we require that, if execution of \( S \) occurs, \( \mathcal{A} \) can not tamper with, or influence, this execution’s outputs. These notions are formalized by the security game in Definition 2.

We note that Definition 2 binds execution of \( S \) to the time between \( \mathcal{H} \) issuing the request and receiving the response. Therefore, if a PoX scheme is secure according to this definition, \( \mathcal{H} \) can be certain about freshness of the execution. In the same vein, the output produced by such execution is also guaranteed to be fresh. This timeliness property is important to avoid replays of previous valid executions; in fact, it is essential for safety-critical applications. See Section 7.3 for examples.

Correctness of the Executable: we stress that the purpose of PoX is to guarantee that \( S \), as specified by \( \mathcal{H} \), was executed. Similar to Trusted Execution Environments targeting high-end CPUs, such as Intel SGX, PoX schemes do not aim to check correctness and absence of implementation bugs in \( S \). As such, it is not concerned with run-time attacks that exploit bugs and vulnerabilities in \( S \) implementation itself, to change its expected behavior (e.g., by executing \( S \) with inputs crafted to exploit \( S \) bugs and hijack its control flow). In particular, correctness of \( S \) need not be assured by the low-end \( \mathcal{P} \). Since \( \mathcal{H} \) is a more powerful device and knows \( S \), it has the ability (and more computational resources) to employ various vulnerability detection methods (e.g., fuzzing [10] or static analysis [14]) or even software formal verification (depending on the level of rigor desired) to avoid or detect implementation bugs in \( S \). This type of techniques can be performed offline before sending \( \mathcal{H} \) to \( \mathcal{P} \) and the whole issue is orthogonal to the PoX functionality. We also note that, if \( \mathcal{H} \) needs to be instrumented for PoX (see Section 5.1 for a discussion on this requirement), it is important to ensure that this instrumentation does not introduce any bugs/vulnerabilities into \( S \).

Physical Attacks: physical and hardware-focused attacks are out of scope of this paper. Specifically, we assume that \( \mathcal{A} \) can not modify code in ROM, induce hardware faults, or retrieve \( \mathcal{P} \) secrets via physical presence side-channels. Protection against such attacks is considered orthogonal and could be supported via standard physical security techniques [38]. This assumption is inline with other hybrid architectures [7, 15, 20, 29].

4.2 MCU Assumptions

We assume the same machine model introduced in VRASED and make no additional assumptions. We review these assumptions throughout the rest of this section and then formalize them as an LTL machine model in Section 6.

Verification of the entire CPU is beyond the scope of this paper. Therefore, we assume the CPU architecture strictly adheres to, and correctly implements, its specifications. In particular, our design and verification rely on the following simple axioms:

A1 – Program Counter (PC): PC always contains the address of the instruction being executed in a given CPU cycle.
Definition 1 (Proof of Execution (PoX) Scheme).

A Proof of Execution (PoX) scheme is a tuple of algorithms \( \{ \text{XRequest, XAtomicExec, XProve, XVerify} \} \) performed between \( \mathcal{P} \) \( \mathcal{V} \) where:

1. \( \text{XRequest}^{\mathcal{V} \rightarrow \mathcal{P}}(s, \cdot) \): is an algorithm executed by \( \mathcal{V} \) which takes as input some software \( s \) (consisting of a list of instructions \( \{ s_1, s_2, ..., s_m \} \)). \( \mathcal{V} \) expects an honest \( \mathcal{P} \) to execute \( s \). \text{XRequest} generates a challenge \( \text{chal} \), and embeds it alongside \( s \), into an output request message asking \( \mathcal{P} \) to execute \( s \), and to prove that such execution took place.

2. \( \text{XAtomicExec}^{\mathcal{P}}(\text{ER}, \cdot) \): an algorithm (with possible hardware-support) that takes as input some executable region \( \text{ER} \) in \( \mathcal{P} \)'s memory, containing a list of instructions \( \{ t_1, t_2, ..., t_m \} \). \( \text{XAtomicExec} \) runs on \( \mathcal{P} \) and is considered successful if: (1) instructions in \( \text{ER} \) are executed from its first instruction, \( t_1 \), and end at its last instruction, \( t_m \); (2) \( \text{ER} \)'s execution is atomic, i.e., if \( E \) is the sequence of instructions executed between \( t_1 \) \( t_m \), then \( \{ e \in E \} \subseteq \text{ER} \); and (3) \( \text{ER} \)'s execution flow is not altered by external events, i.e., MCU interrupts or DMA events. The \( \text{XAtomicExec} \) algorithm outputs result string \( O \). Note that \( O \) may be a default string (\( \perp \)) if \( \text{ER} \)'s execution does not result in any output.

3. \( \text{XProve}^{\mathcal{P} \rightarrow \mathcal{V}}(\text{ER}, \text{chal}, \cdot) \): an algorithm (with possible hardware-support) that takes as input some \( \text{ER} \), \text{chal} and \( O \) and is run by \( \mathcal{P} \) to output \( \mathcal{V} \), i.e., a proof that \( \text{XRequest}^{\mathcal{V} \rightarrow \mathcal{P}}(s, \cdot) \) and \( \text{XAtomicExec}^{\mathcal{P}}(\text{ER}, \cdot) \) happened (in this sequence) and that \( O \) was produced by \( \text{XAtomicExec}^{\mathcal{P}}(\text{ER}, \cdot) \).

4. \( \text{XVerify}^{\mathcal{P} \rightarrow \mathcal{V}}(\mathcal{H}, O, S, \cdot) \): an algorithm executed by \( \mathcal{V} \) with the following inputs: some \( S \), \( \mathcal{H} \), and \( O \). The \( \text{XVerify} \) algorithm checks whether \( \mathcal{H} \) is a valid proof of the execution of \( S \) (i.e., executed memory region \( \text{ER} \) corresponds to \( S \)) on \( \mathcal{P} \) given the challenge \( \text{chal} \), and if \( O \) is an authentic output/result of such an execution. If both checks succeed, \( \text{XVerify} \) outputs 1, otherwise it outputs 0.

Remark: In the parameters list, \( (\cdot) \) denotes that additional parameters might be included, depending on the specific PoX construction.

Definition 2 (PoX Security Game).

- Let \( t_{req} \) denote time when \( \mathcal{V} \) issues \( \text{chal} \leftarrow \text{XRequest}^{\mathcal{V} \rightarrow \mathcal{P}}(s) \).
- Let \( t_{verif} \) denote time when \( \mathcal{V} \) receives \( \mathcal{H} \) and \( O \) back from \( \mathcal{P} \) in response to \( \text{XRequest}^{\mathcal{V} \rightarrow \mathcal{P}}(s, \cdot) \).
- Let \( O \equiv \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \) denote that \( \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \) produces output \( O \). Conversely, \( O \neq \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \) indicates \( O \) is not produced by \( \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \).

2.1 PoX Security Game (PoX game): Challenger plays the following game with \( \text{Adv} \):

1. \( \text{Adv} \) is given full control over \( \mathcal{P} \)'s software state and oracle access to calls to the algorithms \( \text{XAtomicExec}^{\mathcal{P}} \) and \( \text{XProve}^{\mathcal{P}} \).
2. At time \( t_{req} \), \( \text{Adv} \) is presented with software \( S \) and challenge \( \text{chal} \).
3. \( \text{Adv} \) wins in two cases:
   (a) None or incomplete execution: \( \text{Adv} \) produces \( (\mathcal{H}_{\text{adv}}, \mathcal{O}_{\text{adv}}) \), such that \( \text{XVerify}(\mathcal{H}_{\text{adv}}, \mathcal{O}_{\text{adv}}, S, \text{chal}, \cdot) = 1 \), without calling \( \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \).
   (b) Execution with tampered output: \( \text{Adv} \) calls \( \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \) and can produce \( (\mathcal{H}_{\text{adv}}, \mathcal{O}_{\text{adv}}) \), such that \( \text{XVerify}(\mathcal{H}_{\text{adv}}, \mathcal{O}_{\text{adv}}, S, \text{chal}, \cdot) = 1 \) and \( \mathcal{O}_{\text{adv}} \neq \text{XAtomicExec}^{\mathcal{P}}(S, t_{req} \rightarrow t_{verif}) \).

2.2 PoX Security Definition:

A PoX scheme is considered secure for security parameter \( \lambda \) if, for all PPT adversaries \( \text{Adv} \), there exists a negligible function \( \text{negl} \) such that:

\[
P_r(\text{Adv, PoX-game}) \leq \text{negl}(\lambda)
\]

A2 – Memory Address: Whenever memory is read or written, a data-address signal \( (O_{\text{addr}}) \) contains the address of the corresponding memory location. For a read access, a data readable bit \( (R_{\text{en}}) \) must be set, while, for a write access, a data write-enable bit \( (W_{\text{en}}) \) must be set.

A3 – DMA: Whenever the DMA controller attempts to access the main system memory, a DMA-address signal \( (\text{DMA}_{\text{addr}}) \) reflects the address of the memory location being accessed and a DMA-enable bit \( (\text{DMA}_{\text{en}}) \) must be set. DMA cannot access memory when \( \text{DMA}_{\text{en}} \) is off (logical zero).

A4 – MCU Reset: At the end of a successful reset routine, all registers (including \( \text{PC} \)) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified when a reset happens, the corresponding reset signal is set.

The same signal is also set when the MCU initializes for the first time.

A5 – Interrupts: Whenever an interrupt occurs, the corresponding \( \text{irq} \) signal is set.

5 APEX: A Secure PoX Architecture

We now present APEX, a new PoX architecture that realizes the PoX security definition in Definition 2. The key aspect of APEX is a computer-aided formally verified and publicly available implementation thereof. This section first provides some intuition behind APEX’s design. All APEX properties are overviewed informally in this section and are later formalized in Section 6.

In the rest of this section we use the term “unprivileged
Definition 3 (Proof of Execution Protocol). *APEX* instantiates a PoX = (XRequest, XAtomicExec, XProve, XVerify) scheme behaving as follows:

1. XRequest\(^{PoX}\) = \((\mathcal{S}, \mathcal{E}_\text{min}, \mathcal{E}_\text{max}, \mathcal{O}_\text{range}, \mathcal{O}_\text{max})\): includes a set of configuration parameters \(\mathcal{E}_\text{min}\), \(\mathcal{E}_\text{max}\), \(\mathcal{O}_\text{range}\), and \(\mathcal{O}_\text{max}\). The Executable Range (ER) is a contiguous memory block in which \(\mathcal{S}\) is to be installed: \(\mathcal{E} = [\mathcal{E}_\text{min}, \mathcal{E}_\text{max}]\). Similarly, the Output Range (OR) is also configurable and defined by \(\mathcal{O}\)’s request as \(\mathcal{O} = [\mathcal{O}_\text{range}, \mathcal{O}_\text{max}]\). If \(\mathcal{S}\) does not produce any output \(\mathcal{O}_\text{range} = \mathcal{O}_\text{max} = \bot\) \(\mathcal{S}\) is the software to be installed in ER and executed. If \(\mathcal{S}\) is unspecified \((\mathcal{S} = \bot)\) the protocol will execute whatever code was pre-installed on ER on \(\mathcal{Prv}\), i.e., \(\mathcal{V}\) is not required to provide \(\mathcal{S}\) in every request, only when it wants to update ER contents before executing it. If the code for \(\mathcal{S}\) is sent by \(\mathcal{V}\), untrusted auxiliary software in \(\mathcal{Prv}\) is responsible for copying \(\mathcal{S}\) into ER. \(\mathcal{Prv}\) also receives a random 1-bit challenge \(\mathcal{Chal}\) (|\(\mathcal{Chal}\)| = 1) as part of the request, where 1 is the security parameter.

2. XAtomicExec\(^{PoX}\) (ER, OR, METADATA): This algorithm starts with unprivileged auxiliary software writing the values of: \(\mathcal{E}_\text{min}\), \(\mathcal{E}_\text{max}\), \(\mathcal{O}_\text{range}\), \(\mathcal{O}_\text{max}\) and \(\mathcal{Chal}\) to a special pre-defined memory region denoted by METADATA. *APEX*’s verified hardware enforces immutability, atomic execution and access control rules according to the values stored in METADATA; details are described in Section 5.1. Finally, it begins execution of \(\mathcal{S}\) by setting the program counter to the value of \(\mathcal{ER}_\text{min}\).

3. XProve\(^{PoX}\) (ER, \(\mathcal{Chal}\), OR): produces proof of execution \(\mathcal{H}\). \(\mathcal{V}\) allows \(\mathcal{V}\) to decide whether: (1) code contained in ER actually executed; (2) ER contained specified (expected) \(\mathcal{S}\)’s code during execution; (3) this execution is fresh, i.e., performed after the most recent XRequest; and (4) claimed output in OR is indeed produced by this execution. As mentioned earlier, *APEX* uses VRASED’s RA architecture to compute \(\mathcal{H}\) by attesting at least the executable, along with its output, and corresponding execution metadata. More formally:

\[
\mathcal{H} = \text{HMAC}(\text{KDF}(\mathcal{X}, \mathcal{Chal}), \mathcal{ER}, \mathcal{OR}, \text{METADATA},...) \tag{1}
\]

METADATA also contains the EXEC flag that is read-only to all software running in \(\mathcal{Prv}\) and can only be written to by *APEX*’s formally verified hardware. This hardware monitors execution and sets EXEC = 1 only if ER executed successfully (XAtomicExec) and memory regions of METADATA, ER, and OR were not modified between the end of ER’s execution and the computation of \(\mathcal{H}\). The reasons for these requirements are detailed in Section 5.2. If any malware residing on \(\mathcal{Prv}\) attempts to violate any of these properties *APEX*’s verified hardware (provably) sets EXEC to zero. After computing \(\mathcal{H}\), \(\mathcal{Prv}\) returns it and contents of OR (\(\mathcal{O}\)) produced by ER’s execution to \(\mathcal{V}\).

4. XVerify\(^{PoX}\) (\(\mathcal{H}\), \(\mathcal{O}\), \(\mathcal{S}\), METADATA\(_\text{rf}\)): Upon receiving \(\mathcal{H}\) and \(\mathcal{O}\), \(\mathcal{V}\) checks whether \(\mathcal{H}\) is produced by a legitimate execution of \(\mathcal{S}\) and reflects parameters specified in XRequest, i.e., \(\mathcal{METADATA}_{\text{rf}} = \mathcal{Chal}||\mathcal{O}_\text{range}||\mathcal{O}_\text{max}||\mathcal{ER}_\text{min}||\mathcal{ER}_\text{max}||\text{EXEC} = 1\). This way, \(\mathcal{V}\) concludes that \(\mathcal{S}\) successfully executed on \(\mathcal{Prv}\) and produced output \(\mathcal{O}\) if:

\[
\mathcal{H} = \text{HMAC}(\text{KDF}(\mathcal{X}, \mathcal{Chal}_{\text{rf}}), \mathcal{S}, \mathcal{O}, \mathcal{METADATA}_{\text{rf}})\ldots \tag{2}
\]

Table 1: Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Current Program Counter value</td>
</tr>
<tr>
<td>(R_m)</td>
<td>Signal that indicates if the MCU is reading from memory (1-bit)</td>
</tr>
<tr>
<td>(W_m)</td>
<td>Signal that indicates if the MCU is writing to memory (1-bit)</td>
</tr>
<tr>
<td>(D_{\text{addr}})</td>
<td>Address for an MCU memory access</td>
</tr>
<tr>
<td>(D_{\text{valid}})</td>
<td>Signal that indicates if DMA is currently enabled (1-bit)</td>
</tr>
<tr>
<td>(D_{\text{addr}})</td>
<td>Memory address being accessed by DMA, if any</td>
</tr>
<tr>
<td>(\text{en})</td>
<td>Signal that indicates if an interrupt is happening</td>
</tr>
<tr>
<td>(\text{hal})</td>
<td>Memory region where SW-Att is stored: (CR = [\mathcal{ER}<em>\text{min}, \mathcal{ER}</em>\text{max}])</td>
</tr>
<tr>
<td>(\text{MR})</td>
<td>MAC Region Memory region in which SW-Att computation result is written: (MR = [\mathcal{MR}<em>\text{min}, \mathcal{MR}</em>\text{max}])</td>
</tr>
<tr>
<td>(\mathcal{AR})</td>
<td>Attestation Region Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from (\mathcal{V}); AR = ([\mathcal{AR}<em>\text{min}, \mathcal{AR}</em>\text{max}])</td>
</tr>
<tr>
<td>(\mathcal{ER})</td>
<td>Key Region Memory region that stores (\mathcal{X})</td>
</tr>
<tr>
<td>(\mathcal{XS})</td>
<td>Exclusive Stack Region Exclusive memory region that contains SW-Att’s stack and can be only accessed by SW-Att</td>
</tr>
<tr>
<td>(\text{reset})</td>
<td>A 1-bit signal that boots/reset the MCU when set to logical 1</td>
</tr>
<tr>
<td>(\mathcal{ER})</td>
<td>Execution Region Memory region that stores an executable to be executed: (ER = [\mathcal{ER}<em>\text{min}, \mathcal{ER}</em>\text{max}])</td>
</tr>
<tr>
<td>(\mathcal{OR})</td>
<td>Output Region Memory region that stores execution output: (OR = [\mathcal{OR}<em>\text{min}, \mathcal{OR}</em>\text{max}])</td>
</tr>
<tr>
<td>(\text{EXEC})</td>
<td>1-bit execution flag indicating whether a successful execution has happened</td>
</tr>
<tr>
<td>(\text{METADATA})</td>
<td>Memory region containing APEX’s metadata</td>
</tr>
</tbody>
</table>

To VRASED’s implementation of SW-Att (see Section 3) which is formally verified and can not be modified by \(\mathcal{Adv}\), since it is stored in ROM. *APEX* is designed such that no changes to SW-Att are required. Therefore, both functionalities (RA and PoX, i.e., VRASED and *APEX*) can co-exist on the same device without interfering with each other.

Notation is summarized in Table 1.

5.1 Protocol and Architecture

![Figure 1: Overview of APEX’s workflow](image-url)

APEX implements a secure PoX = (XRequest, XAtomicExec, XProve, XVerify) scheme conforming to...
Definition 3. The steps in APEX workflow are illustrated in Figure 1. The main idea is to first execute code contained in ER. Then, at some later time, APEX invokes VRASED verified RA functionality to attest the code in ER and include, in the attestation result, additional information that allows \( V^\text{rf} \) to verify that ER code actually executed. If ER execution produces an output (e.g., \( P_{\text{rv}} \) is a sensor running ER’s code to obtain some physical/ambient quantity), authenticity and integrity of this output can also be verified. That is achieved by including the EXEC flag among inputs to HMAC computed as part of VRASED RA. The value of this flag is controlled by APEX formally verified hardware and its memory can not be written by any software running on \( P_{\text{rv}} \). APEX hardware module runs in parallel with the MCU, monitoring its behavior and deciding the value of EXEC accordingly.

Figure 2 depicts APEX’s architecture. In addition to VRASED hardware that provides secure RA by monitoring a set of CPU signals (see Section 3.2), APEX monitors values stored in the dedicated physical memory region called METADATA. METADATA contains addresses/pointers to memory boundaries of ER (i.e., \( E_{\text{min}} \) and \( E_{\text{max}} \)) and memory boundaries of expected output: \( O_{\text{min}} \) and \( O_{\text{max}} \). These addresses are sent by \( V^\text{rf} \) as part of XRequest, and are configurable at run-time. The code \( S \) to be stored in ER is optionally\(^2\) sent by \( V^\text{rf} \).

METADATA includes the EXEC flag, which is initialized to 0 and only changes from 0 to 1 (by APEX’s hardware) when ER execution starts, i.e., when the PC points to \( E_{\text{min}} \). Afterwards, any violation of APEX’s security properties (detailed in Section 5.2) immediately changes EXEC back to 0. After a violation, the only way to set the flag back to 1 is to re-start execution of ER from the very beginning, i.e., with PC=\( E_{\text{min}} \). In other words, APEX verified hardware makes sure that EXEC value covered by the HMAC’s result (represented by \( \psi^\text{rf} \)) is 1, if and only if ER code executed successfully. As mentioned earlier, we consider an execution to be successful if it runs atomically (i.e., without being interrupted), from its first \( E_{\text{min}} \) to its last instruction \( E_{\text{max}} \).

In addition to EXEC, HMAC covers a set of parameters (in METADATA memory region) that allows \( V^\text{rf} \) to check whether executed software was indeed located in ER = [\( E_{\text{min}}, E_{\text{max}} \)]. If any output is expected, \( V^\text{rf} \) specifies a memory range \( \text{OR} = [O_{\text{min}}, O_{\text{max}}] \) for storing output. Contents of \( \text{OR} \) are also covered by the computed HMAC, allowing \( V^\text{rf} \) to verify authenticity of the output of the execution.

**Remark:** Our notion of successful execution requires \( S \) to have a single exit point \(-E_{\text{max}}\). Any self-contained code with multiple legal exits can be trivially instrumented to have a single exit point by replacing each exit instruction with a jump to the unified exit point \( E_{\text{max}} \). This notion also requires \( S \) to run atomically. Since this constraint might be undesirable for some real-time systems, we discuss how to relax it in Section 8.

---

\( ^2 \) Sending the code to be executed is optional because \( S \) might be pre-installed on \( P_{\text{rv}} \). In that case the proof of execution will allow \( V^\text{rf} \) to conclude that the pre-installed \( S \) was not modified and that it was executed.
we rely on VRASED with XAtomicExec. They are written into metadata-dedicated physical addresses MP1 - Executable/Output (5.2.2 Metadata Protection: that it produced output OR Response Protection: MP2 - Ephemeral Atomicity: APEX must ensure that the attestation result reflects the code that executed. Lack of this property would allow Adv to overwrite some other code ER_adv, overwrite it with expected ER and finally call XProve. This would result in a valid proof of execution of ER even though ER_adv was executed instead.

5.2.2 Metadata Protection:

MP1 - Executable/Output (ER/OR) Boundaries: APEX hardware ensures properties EP1, EP2, and EP3 according to values: ER_min, ER_max, OR_min, OR_max. These values are configurable and can be decided by Vrf based on application needs. They are written into metadata-dedicated physical addresses in \( \mathbb{P}_{\text{rv}} \) memory before ER execution. Therefore, once ER execution starts, APEX hardware must ensure that such values remain unchanged until XProve completes. Otherwise, Adv could generate valid attestation results, by attesting \( \{ER_{\min}, ER_{\max}\} \), while, in fact, having executed code in a different region: \( \{ER_{\min}^{adv}, ER_{\max}^{adv}\} \).

MP2 - Response Protection: The appropriate response to Vrf’s challenge must be unforgeable and non-invertible. Therefore, in the XProve routine, \( X_r \) used to compute HMAC must never be leaked (with non-negligible probability) and HMAC implementation must be functionally correct, i.e., adhere to its cryptographic specification. Moreover, contents of memory being attested must not change during HMAC computation. We rely on VRASED to ensure these properties. Also, to ensure trustworthiness of the response, APEX guarantees that no software in \( \mathbb{P}_{\text{rv}} \) can ever modify EXEC flag and that, once EXEC = 0, it can only become 1 if ER’s execution re-starts afresh.

5.2.3 Challenge Temporal Consistency: APEX must ensure that Chal cannot be modified between ER’s execution and HMAC computation in XProve. Without this property, the following attack is possible: (1) \( \mathbb{P}_{\text{rv}} \)-resident malware executes ER properly (i.e., by not violating EP1-EP3 and MP1-MP2), resulting in EXEC = 1 after execution stops, and (2) at a later time, malware receives Chal from Vrf and simply calls XProve on this Chal without executing ER. As a result, malware would acquire a valid proof of execution (since EXEC remains 1 when the proof is generated) even though no ER execution occurred before Chal was received. Such attacks are prevented by setting EXEC = 0 whenever the memory region storing Chal is modified.

6 Formal Specification & Verified Implementation

Our formal verification approach starts by formalizing APEX sub-properties Linear Temporal Logic (LTL) to define invariants that must hold throughout the MCU operation. We then use a theorem prover [18] to write a computer-aided proof that the conjunction of the LTL sub-properties imply an end-to-end formal definition for the guarantee expected from APEX hardware. APEX correctness, when properly composed with VRASED guarantees, yields a PoX scheme secure according to Definition 2. This is proved by showing that, if the composition between the two is implemented as described in Definition 3, VRASED security can be reduced to APEX security.

APEX hardware module is composed of several sub-modules written in Verilog Hardware Description Language (HDL). Each sub-module is responsible for enforcing a set of LTL sub-properties and is described as an FSM in Verilog at Register Transfer Level (RTL). Individual sub-modules are combined into a single Verilog design. The resulting composition is converted to the SMV model checking language using the automatic translation tool Verilog2SMV [26]. The resulting SMV is simultaneously verified against all LTL specifications, using the model checker NuSMV [12], to prove that the final Verilog of APEX complies with all necessary properties.

6.1 Machine Model

Definition 4 models, in LTL, the behavior of low-end MCUs considered in this work. It consists of a subset of the machine model introduced by VRASED. Nonetheless, this subset models all MCU behavior relevant for stating and verifying correctness of APEX’s implementation. Modify_Mem models that a given memory address can be modified by a CPU instruction or by a DMA access. In the former, W_en signal must be set and D_addr must contain the target memory address. In the latter, DMA_en signal must be
Definition 4. Machine Model (subset)

1. Modify_Mem(i) \rightarrow (W_{en} \land D_{addr} = i) \lor (DMA_{en} \land DMA_{addr} = i)
2. Interrupt \rightarrow irq
3. MR, CR, AR, KR, XS, and METADATA are non-overlapping memory regions

set and DMA_{addr} must contain the target DMA address. The requirements for reading from a memory address are similar, except that instead of W_{en}, R_{en} must be on. We do not explicitly state this behavior since it is not used in APEX proofs. For the same reason, modeling the effects of instructions that only modify register values (e.g., ALU operations, such as add and mul) is also not necessary. The machine model also captures the fact that, when an interrupt happens during execution, the irq signal in MCU hardware is set to 1.

With respect to memory layout, the model states that MR, CR, AR, KR, XS, and METADATA are disjoint memory regions. The first five memory regions are defined in VRASED. As shown in Figure 2, METADATA is a fixed memory region used by APEX to store information about software execution status.

6.2 Security & Implementation Correctness

We use a two-part strategy to prove that APEX is a secure PoX architecture, according to Definition 2:

[A]: We show that properties EP1-EP3 and MP1-MP3, discussed in Section 5.2 and formally specified next in Section 6.3, are sufficient to guarantee that EXEC flag is 1 iff S indeed executed on Prev. To show this, we compose a computer proof using SPOT LTL proof assistant [18].

[B]: We use cryptographic reduction proofs to show that, as long as part A holds, VRASED security can be reduced to APEX’s PoX security from Definition 2. In turn, HMAC’s existential unforgeability can be reduced to VRASED’s security [15]. Therefore, both APEX and VRASED rely on the assumption that HMAC is a secure MAC.

In the rest of this section, we convey the intuition behind both of these steps. Proof details are in Appendix B.

The goal of part A is to show that APEX’s sub-properties imply Definition 5. LTL specification in Definition 5 captures the conditions that must hold in order for EXEC to be set to 1 during execution of XProve, enabling generation of a valid proof of execution. This specification ensures that, in order to have EXEC = 1 during execution of XProve (i.e., for \( [EXEC \land PC \in CR] \) to hold), at least once before such time the following must have happened:

1. The system reached state S_0 where software stored in ER started executing from its first instruction (PC = ER_{min}).
2. The system eventually reached a state S_1 when ER finished executing (PC = ER_{max}). In the interval between S_0 and S_1, PC kept executing instructions within ER, there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S_2 when XProve started executing (PC = CR_{min}). In the interval between S_0 and S_2, METADATA and ER regions were not modified.
4. In the interval between S_0 and S_2, OR region was only modified by ER’s execution, i.e., PC \in ER \lor \neg Modify_Mem(OR).

Figure 3 shows the time windows wherein each memory region must not change during APEX’s PoX as implied by APEX’s correctness (Definition 5). Violating any of these conditions will cause EXEC have value 0 during XProve’s computation. Consequently, any violation will result in \( \forall \mathcal{H} \) rejecting the proof of execution since it will not conform to the expected value of \( \mathcal{H} \), per Equation 2 in Definition 3.

The intuition behind the cryptographic reduction (part B) is that computing \( \mathcal{H} \) involves simply invoking VRASED SW-Att with MR = Chal, ER \in AR, OR \in AR, and METADATA \in AR. Therefore, a successful forgery of APEX’s \( \mathcal{H} \) implies breaking VRASED security. Since \( \mathcal{H} \) always includes the value of EXEC, this implies that APEX is PoX-secure (Definition 2). The complete reduction is presented in Appendix B.

6.3 APEX’s Sub-Properties in LTL

We formalize the necessary sub-properties enforced by APEX as LTL specifications 3–12 in Definition 6. We describe how they map to high-level notions EP1-EP3 and MP1-MP3 discussed in Section 5.2. Appendix B discusses a computer proof that the conjunction of this set of properties is sufficient to satisfy a formal definition of APEX correctness from Definition 5.

LTL 3 enforces EP1 – Ephemeral immutability by making sure that whenever ER memory region is written by either CPU or DMA, EXEC is immediately set to logical 0 (false).

EP2 – Ephemeral Atomicity is enforced by a set of three LTL specifications. LTL 4 enforces that the only way for ER’s execution to terminate, without setting EXEC to logical 0, is through its last instruction: PC = ER_{max}. This is specified by checking the relation between current and next PC values using LTL next operator. In particular, if current PC value is within ER, and next PC value is outside SW-Att region, then either current PC value is the address of ER_{max}, or EXEC is set to 0 in the next cycle. Also, LTL 5 enforces 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 making sure that ER always executes in its entirety. Finally, LTL 6 enforces that EXEC is set to zero if an interrupt happens in the middle of ER execution. Even though LTLs 4 and 5 already enforce that PC can not change to anywhere outside ER, interrupts could be programmed to return to an arbitrary instruction within ER. Although this would not violate LTLs 4 and 5, it would still modify ER’s behavior. Therefore, LTL 6 is needed to prevent that.

EP3 – Output Protection is enforced by LTL 7 by making sure that: (1) DMA controller does not write into OR; (2) CPU
Figure 3: Illustration of time intervals that each memory region must remain unchanged in order to produce a valid \( \mathcal{H} \) (\( \text{EXEC} = 1 \)). \( t(X) \) denotes the time when \( PC = X \).

can only modify \( OR \) when executing instructions within \( ER \); and 3) DMA can not be active during \( ER \) execution; otherwise, a compromised DMA could change intermediate results of \( ER \) computation in data memory, potentially modifying \( ER \) behavior.

Similar to EP3, MP1 – Executable/Output Boundaries and MP3 – Challenge Temporal Consistency are enforced by LTL 10. Since \( \text{Chal}_{\text{mem}} \) as well as \( ER_{\text{min}}, ER_{\text{max}}, OR_{\text{min}}, \) and \( OR_{\text{max}} \) are all stored in \( \text{METADATA} \) reserved memory region, it suffices to ensure that \( \text{EXEC} \) is set to logical 0 whenever this region is modified. Also, LTL 8 enforces that \( \text{EXEC} \) is only set to one if \( ER \) and \( OR \) are configured (by \( \text{METADATA} \) values \( ER_{\text{min}}, ER_{\text{max}}, OR_{\text{min}}, OR_{\text{max}} \)) as valid memory regions.

Finally, LTLs 11, and 12 (in addition to \( \text{VRASED} \) verified RA architecture) are responsible for ensuring MP2- Response Protection by making sure that \( \text{EXEC} \) always reflects what is intended by \( \text{APEX} \) hardware. LTL 7 specifies that the only way to change \( \text{EXEC} \) from 0 to 1 is by starting \( ER \)'s execution over. Finally, LTL 12 states that, whenever a reset happens (this also includes the system initial booting state) and execution is initialized, the initial value of \( \text{EXEC} \) is 0. To conclude, recall that \( \text{EXEC} \) is read-only to all software running on \( \text{Prv} \). Therefore, malware can not change it directly.

\( \text{APEX} \) is designed as a set of seven hardware sub-modules, each verified to enforce a subset of properties discussed in this
APEX implementation uses OpenMSP430 [23] as its open source hardware overhead as compared to unmodified OpenMSP430 [23] 44 registers and 302 Look-Up Tables (LUTs) to the underlying ASIC on a larger scale. XProve an attestation token (located inside METADATA) can be accessed in a pre-defined memory address via standard peripheral memory access. We also ensure that EXEC (located inside METADATA) is unmodifiable in software by removing software-write wires in hardware. Finally, as a proof of concept, we use Xilinx Vivado to synthesize an RTL description of the modified HW-Mod and deploy it on the Artix-7 FPGA class. 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) on a larger scale.

7 Implementation & Evaluation

APEX implementation uses OpenMSP430 [23] as its open core implementation. We implement the hardware architecture shown in Figure 2. In addition to APEX and VRASED modules in HW-Mod, we implement a peripheral module responsible for storing and maintaining APEX METADATA. As a peripheral, contents of METADATA can be accessed in a pre-defined memory address via standard peripheral memory access. We also ensure that EXEC (located inside METADATA) is unmodifiable in software by removing software-write wires in hardware. Finally, as a proof of concept, we use Xilinx Vivado to synthesize an RTL description of the modified HW-Mod and deploy it on the Artix-7 FPGA class. 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) on a larger scale.

7.1 Evaluation Results

Hardware & Memory Overhead. Table 2 reports APEX hardware overhead as compared to unmodified OpenMSP430 [23] and VRASED [15]. Similar to the related work [15–17, 44], we consider the hardware overhead in terms of additional LUTs and registers. The increase in the number of LUTs 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 memory overhead required by the sequential logic in APEX FSMs. APEX hardware overhead is small compared to the baseline VRASED; it requires 2% and 12% additional registers and LUTs, respectively. In absolute numbers, it adds 44 registers and 302 Look-Up Tables (LUTs) to the underlying MCU. In terms of memory, APEX needs 9 extra bytes of RAM for storing METADATA. This overhead corresponds to 0.01% of MSP430 16-bit address space.

Run-time. We do not observe any overhead for software’s execution time on the APEX-enabled Pvtv since APEX does not introduce new instructions or modifications to the MSP430 ISA. APEX hardware runs in parallel with the original MSP430 CPU. Run-time to produce a proof of S execution includes: (1) time to execute S (XAtomicExec), and (2) time to compute an attestation token (XProve). The former only depends on S behavior itself (e.g., SW-Att can be a small sequence of instructions or have long loops). As mentioned earlier, APEX does not affect S run time. XProve’s run-time is linear in the size of ER + OR. In the worst-case scenario where these regions occupy the entire program 8kBytes memory, XProve takes around 900ms on an 8MHz device.

Verification Cost. We verify APEX on an Ubuntu 16.04 machine running at 3.40GHz. Results are shown in Table 2. APEX verification requires checking 10 extra invariants (shown in Definition 6) in addition to existing VRASED invariants. It also consumes significantly higher run-time and memory usage than VRASED verification. This is because additional invariants introduce five additional variables (ER, min, ER, max, OR, min, OR, max and EXEC), potentially resulting in an exponential increase in complexity of the model checking process. Nonetheless, the overall verification process is still reasonable for a commodity desktop – it takes around 3 minutes and consumes 280MB of memory.

7.2 Comparison with CFA

To the best of our knowledge, APEX is the first of its kind and thus there are no other directly comparable PoX architectures. However, to provide a (performance and overhead) point of reference and a comparison, we contrast APEX overhead with that state-of-the-art CFA architectures. As discussed in Section 2, even though CFA is not directly applicable for producing proofs of execution with authenticated outputs, we consider it to be the closest-related service, since it reports on the exact execution path of a program.

We consider three recent CFA architectures: Atrium [44], LiteHAx [16], and LO-FAT [17]. Figure 4.a compares APEX to these architectures in terms of number of additional LUTs. In this figure, the black dashed line represents the total cost of the MSP430 MCU: 1904 LUTs. Figure 4.b presents a similar comparison for the amount of additional registers required by these architectures. In this case, the total cost of the MSP430 MCU itself is of 691 registers. Finally, Figure 4.c presents the amount of dedicated RAM required by these architectures (APEX’s dedicated RAM corresponds to the exclusive access stack implemented by VRASED). As expected, APEX incurs much lower overhead. According to our results, the cheapest CFA architecture, LiteHAx, would entail an overhead of nearly 100% LUTs and 300% registers, on MSP430. In addition, LiteHAx would require 150 kB of dedicated RAM. This amount far exceeds entire addressable memory (64 kB) of 16-bit processors, such as MSP430. Results support our claim that CFA is not applicable to this class of low-

<table>
<thead>
<tr>
<th>Hardware Reg LUT</th>
<th>Reserved RAM (bytes)</th>
<th># LTL Invariants</th>
<th>Verification</th>
<th>Time (s)</th>
<th>Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMSP430 [23]</td>
<td>691 1904</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VRASED [15]</td>
<td>721 1964</td>
<td>2332</td>
<td>10</td>
<td>481</td>
<td>0.4</td>
</tr>
<tr>
<td>APEX +VRASED</td>
<td>735 2206</td>
<td>2341</td>
<td>20</td>
<td>1385</td>
<td>183.6</td>
</tr>
</tbody>
</table>

Table 2: Evaluation results.
end devices. Meanwhile, APEX needs a total of 12% additional LUTs and 2% additional registers. VRASED requires about 2 kB of reserved RAM, which is not increased by APEX PoX support.

7.3 Proof of Concept: Authenticated Sensing and Actuation

As discussed in Section 1 an important functionality attainable with PoX is authenticated sensing/actuation. In this section, we demonstrate how to use APEX to build sensors and actuators that “can not lie”.

As a running example we use a fire sensor: a safety-critical low-end embedded device commonly present in households and workplaces. It consists of an MCU equipped with analog hardware for measuring physical/chemical quantities, e.g., temperature, humidity, and CO₂ level. It is also usually equipped with actuation-capable analog hardware, such as a buzzer. Analog hardware components are directly connected to MCU General Purpose Input/Output (GPIO) ports. GPIO ports are physical wires directly mapped to fixed memory locations in MCU memory. Therefore, software running on the MCU can read physical quantities directly from GPIO memory.

In this example, we consider that MCU software periodically reads these values and transmits them to a remote safety authority, e.g., a fire department, which then decides whether to take action. The MCU also triggers the buzzer actuator whenever sensed values indicate a fire. Given the safety-critical nature of this application, the safety authority must be assured that reported values are authentic and were produced by execution of expected software. Otherwise, malware could spoof such values (e.g., by not reading them from the proper GPIO). PoX guarantees that reported values were read from the correct GPIO port (since the memory address is specified by instructions in the ER executable), and produced output (stored in OR) was indeed generated by execution of ER and was unmodified thereafter. Thus, upon receiving sensed values accompanied by a PoX, the safety authority is assured that the reported sensed value can be trusted.

As a proof of concept, we use APEX to implement a simple fire sensor that operates with temperature and humidity quantities. It communicates with a remote \( \Psi rf \) (e.g., the fire department) using a low-power ZigBee radio\(^3\) typically used by low-end CPS/IoT devices. Temperature and humidity analog devices are connected to a APEX-enabled MSP430 MCU running at 8MHz and synthesized using a Basys3 Artix-7 FPGA board. As shown in Figure 5, MCU GPIO ports connected to the temperature/humidity sensor and to the buzzer. APEX is used to prove execution of the fire sensor software. This software is shown in Figure 8a in Appendix C. It consists of two main functions: ReadSensor and SoundAlarm. Proofs of execution are requested by the safety authority via XRequest to issue commands to execute these functions. ReadSensor reads and processes the value generated by temperature/humidity analog device memory-mapped GPIO, and copies this value to OR. The SoundAlarm function turns the buzzer on for 2 seconds, i.e., it writes “1” to the memory address mapped to the buzzer, busy-waits for 2 seconds, and then writes “0” to the same memory location. This implementation corresponds to the one in the open-source repository \(^4\) and was ported to a APEX-enabled MCU. The porting effort was minimal: it involved around 30 additional lines of C code, mainly for re-implementing sub-functions originally implemented as shared APIs, e.g., digitalRead/Write. Finally, we transformed ported code to be compatible with APEX’s PoX architecture. Details can be found in Appendix C.

8 Limitations & Future Directions

In the following we discuss some limitations in APEX current approach and directions for future work.

Shared libraries. In order to produce a valid proof, \( \Psi rf \) must ensure that execution of \( \hat{S} \) does not depend on external code located outside the executable range \( ER \) (e.g., shared libraries). A call to such code would violate LTL 4, resulting in \( EXEC = 0 \) during the attestation computation. To support this type of executable one can transform it into a self-contained executable by statically linking all dependencies during the compilation time.

Self-modifying code (SMC). SMC is a type of executable that alters itself while executing. Clearly, this executable type violates LTL 3 that requires code in \( ER \) to remain unchanged during \( ER \)'s execution. It is unclear how APEX can be adapted to support SMC; however, we are unaware of any legitimate and realistic use-case of SMC in our targeted platforms.

Atomic Execution & Interrupts. The notion of successful execution, defined in Section 5.1, prohibits interruptions during \( \hat{S} \)'s execution. This limitation can be problematic especially on systems with strict real-time constraints. In this case, the PoX executable might be interrupted by a higher priority task and, in order to provide a valid proof of execution, execution must start over. On the other hand, simply resuming \( \hat{S} \) execution after an interrupt may result in attacks where malware modifies intermediate execution results, in data memory, consequently influencing the correctness of this execution’s output. One possible way to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) themselves immutable and uninterruptible from the start of execution until the end of attestation; and (2) included in the attested memory range during the attestation process. \( \Psi rf \) could then use the PoX result \( \mathcal{H} \) to determine whether an interrupt that may have happened during the execution is malicious. This idea needs to be examined carefully, because even the accurate definition of PoX correctness and security in this case becomes challenging.

\(^{3}\)https://www.zigbee.org/

\(^{4}\)https://github.com/Seeed-Studio/LaunchPad_Kit
Figure 4: Overhead comparison between APEX and CFA architectures. Dashed lines represent total hardware cost of MSP430.

Figure 5: Hardware setup for a fire sensor using APEX

9 Conclusion

This paper introduces APEX, a novel and formally verified security service targeting low-end embedded devices. It allows a remote untrusted prover to generate unforgeable proofs of remote software execution. We envision APEX’s use in many IoT application domains, such as authenticated sensing and actuation. Our implementation of APEX is realized on a real embedded system platform, MSP430, synthesized on an FPGA, and the verified implementation is publicly available. Our evaluation shows that APEX has low overhead for both hardware footprint and time for generating proofs of execution.

Acknowledgements: The authors thank the designated shepherd (Dr. Sven Bugiel) for his guidance, and the anonymous reviewers for their valuable feedback. UC Irvine authors’ work was supported by Army Research Office (ARO), under contract W911NF-16-1-0536 and Semiconductor Research Corporation (SRC), under contract 2019-TS-2907.

References

APPENDIX

A Sub-Module Verification

APEX is designed as a set of seven sub-modules. We now describe APEX’s verified implementation, by focusing on two of these sub-modules and their corresponding properties. The Verilog implementation of omitted sub-modules is available in [1]. Each sub-module enforces a sub-set of the LTL specifications in Definition 6. As discussed in Section 6, sub-modules are designed as FSMs. In particular, we implement them as Mealy FSMs, i.e., their output changes as a function of both the current state and current input values. Each FSM takes as input a subset of the signals shown in Figure 2 and produces only one output – EXEC – indicating violations of PoX properties.

To simplify the presentation, we do not explicitly represent the value of EXEC for each state transition. Instead, we define the following implicit representation:

1. EXEC is 0 whenever an FSM transitions to NotExec state.
2. EXEC remains 0 until a transition leaving NotExec state is triggered.
3. EXEC is 1 in all other states.
4. Sub-modules composition: Since all PoX properties must simultaneously hold, the value of EXEC produced by APEX is the conjunction (logical AND) of all sub-modules’ individual EXEC flags.

Figure 7: Verified FSM for LTL 10, a.k.a., MP3- Challenge Temporal Consistency.

The only possible path from notER to midER is through f stER. Similarly, the only path from midER to notER is through lstER. A transition to the NotExec state is triggered whenever: (1) any sequence of values for PC do not follow the aforementioned conditions, or (2) irq is logical 1 while PC is inside ER. Lastly, the only way to transition out of the NotExec state is to restart ER’s execution.

Figure 7 shows the FSM verified to comply with LTL 10 (MP3- Challenge Temporal Consistency). 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 METADATA is modified in software. It transitions back to Run when ER’s execution is restarted without such violation.

B Proofs of Implementation Correctness & Security

In this section we discuss the computer proof for APEX’s implementation correctness (Theorem 1) and the reduction, showing that APEX is a secure PoX architecture as long as VRASED is a secure RA architecture (Theorem 2). A formal LTL computer

\[ \text{Theorem 1. Definition 4 \& LTLs } 3 \rightarrow 12 \rightarrow \text{Definition 5.} \]

proof for Theorem 1 is available at [1]. We here discuss the intuition behind such proof. Theorem 1 states that LTLs 3 – 12, when considered in conjunction with the machine model in Definition 4, imply APEX’s implementation correctness.

Recall that Definition 5 states that, in order to have EXEC = 1 during the computation of XProve, at least once before such event (EXEC = 1) the following must have happened:

1. The system reached state S0 in which the software stored in ER started executing from its first instruction (PC = ERmin).
2. The system eventually reached a state S1 when ER finished executing (PC = ERmax). In the interval between S0 and S1 PC remained executing instructions within ER, and there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S2 when XProve started executing (PC = CRmin). In the interval between

Figure 6: Verified FSM for LTLs 4-6, a.k.a., EP2- Ephemeral Atomicity.

Figure 6 represents a verified model enforcing LTLs 4-6, corresponding to the high-level property EP2- Ephemeral Atomicity. The FSM consists of five states, notER and midER represent states when PC is: (1) outside ER, and (2) within ER respectively, excluding the first (ERmin) and last (ERmax) instructions. Meanwhile, f stER and lstER correspond to states when PC points to the first and last instructions, respectively.
Theorem 1. To comply with conditions 3.a and 3.b in Definition 2, this must be done in either of the following two ways:

1. Provide (negligible probability). Therefore, by contradiction, nonexistence of VRASED can be used to construct (in a polynomial number of steps) (Theorem 2)

2. The code in EXEC (PC ∈ CR) without leaving ER, because LTL 9 guarantees that ER and CR do not overlap, or EXEC = 0.

So far we have argued that to have a token H that reflects EXEC = 1 the code contained in ER must have executed successfully. What remains to be shown is: producing this token implies the code in ER and METADATA are not modified in the interval between S0 and S2 and only ER’s execution can modify OR in the same time interval.

Clearly, the contents of ER can not be modified after S0 because Modify_Mem(ER) directly implies that LTL 3 will set EXEC = 0. The same reasoning is applicable for modifications to METADATA region with respect to LTL 10. The same argument applies to modifying OR, with the only exception that OR modifications are allowed only by the CPU and when PC ∈ ER (LTL 7). This means that OR can only be modified by the execution of ER. In addition, LTL 7 also ensures that DMA is disabled during the execution of ER to prevent unauthorized modification of intermediate results in data memory. Therefore, the timeline presented in Figure 3 is strictly implied by APEX’s implementation. This concludes the reasoning behind Theorem 1.

Proof. (Theorem 2) Assume that AdvPOX is an adversary capable of winning the security game in Definition 2 against APEX with more than negligible probability. We show that, if such AdvPOX exists, then it can be used to construct (in a polynomial number of steps) AdvRA that wins VRASED’s security game (Definition 7) with more than negligible probability. Therefore, by contradiction, nonexistence of AdvRA (i.e., VRASED’s security) implies nonexistence of AdvPOX (APEX’s security).

First we recall that, to win APEX’s security game, AdvPOX must provide (HAdv, OAdv), such that VXVerify(HAdv, OAdv, S, Chal, l) = 1. To comply with conditions 3.a and 3.b in Definition 2, this must be done in either of the following two ways:

Case 1. AdvPOX does not execute S in the time window between treq and tverif (i.e., XAtomicExecPrv(ER, S, treq → tverif)).

Case 2. AdvPOX calls XAtomicExecPrv(S, treq → tverif) but modifies its output O in between the time when the execution of S completes and the time when XProve is called.

However, according to the specification of APEX’s VXVerify algorithm (see Definition 3), a token HAdv will only be accepted if it reflects an input value with EXEC = 1, as expected by Vrf. In APEX’s implementation, O is stored in region OR and S in region ER. Moreover, given Theorem 1, we know that having EXEC = 1 during XProve implies three conditions have been fulfilled:

Cond1. The code in ER executed successfully.

Cond2. The code in ER and METADATA were not modified after starting ER’s execution and before calling XProve.

Cond3. Outputs in OR were not modified after completing ER’s execution and before calling XProve.

The third condition rules out the possibility of Case 2 since that case assumes Adv can modify O, resided in OR, after ER execution and EXEC stays logical 1 during XProve. We further break down Case 1 into three sub-cases:

Case 1.1 AdvPOX does not follow Cond1-Cond3. The only way for AdvPOX to produce (HAdv, OAdv) in this case is to call XProve and directly guess H.

Case 1.2 AdvPOX follows Cond1-Cond3 but does not execute S between treq and tverif. Instead, it produces (HAdv, OAdv) by calling:

\[ O_{Adv} \equiv XAtomicExecPrv(ER_{Adv}, O_{Adv}, S, Chal, l) = 1 \]

Case 1.3 Similar to Case 1.2, with ERAdv being the same region specified by Vrf on XRequest, but instead containing a different executable SAdv.

We show that an adversary that succeeds in any of these cases can be used win VRASED’s security game. To see why this is the case, we note that APEX’s XProve function is implemented by using VRASED’s SW-Att. SW-Att covers memory regions MR (challenge memory) and AR (attested region). Hence, APEX instantiates these memory regions as:
A Fire Sensor’s code written in C

(b) Linker script

Figure 8: Code snippets for (a) fire sensor described in Section 7.3 (b) linker script

C Software Transformation

Recall that the notion of successful execution (in Section 5.1) requires the executable’s entry point to be at the first instruction in ER and the exit point to be at the last instruction in ER. In this section, we discuss how to efficiently transform arbitrary software to conform with this requirement.

Lines 10-17 of Figure 8.a show a (partial) implementation of the ReadSensor function described in Section 7.3. This implementation, when converted to an executable, does not meet APEX’s executable requirement, since the compiler may choose to place one of its sub-functions (instead of ReadSensor) as the entry and/or exit points of the executable. One way to fix this issue is to implement all of its sub-functions as inline functions. However, this may be inefficient; in this example, it would duplicate the code of the same sub-functions (e.g., digitalWrite) inside the executable.

Instead, we create dedicated functions for entry (Line 1-4) and exit (Line 6-8) points, and assign those functions to separate executable sections: “.exec.entry” for the entry and “.exec.exit” for the exit. Then, we label all sub-functions used by ReadSensor as well as ReadSensor itself to the same section – “.exec.body” – and modify the MSP430 linker to place “.exec.body” between “.exec.entry” and “.exec.exit” sections. The modified linker script is shown in Figure 8.b.