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}, and Gene Tsudik\textsuperscript{1}

\textsuperscript{1}University of California, Irvine
\textsuperscript{2}SRI International
\{ivanoliv, nrattana, gene.tsudik\}@uci.edu, karim.eldefrawy@sri.com

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 \textit{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 be equipped with small anemic CPU, 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 could be modified and/or compromised?

In this paper we answer these questions by designing, showing security of, and formally verifying, VAPE: Verified Architecture for Proofs of 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, to authenticated sensing and trustworthy actuation, which are increasingly relevant in the context of safety-critical systems. VAPE architecture is publicly available and our evaluation indicates that it incurs low overhead, affordable even for very low-end embedded devices, e.g., those based on TI MSP430 or AVR ATmega processors.

1 Introduction

The number and diversity of special-purpose computing devices has been increasing dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets, utilized in various “smart” or instrumented settings, such as homes, offices, factories, automotive systems and public venues. Tasks performed by these devices are often safety-critical. For example, a typical industrial control system depends on physical measurements (e.g., 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. 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 infestations 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 \textit{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 \textbf{Proofs of Execution (PoX)} are both important and necessary for securing low-end MCUs. Specifically, we demonstrate in Section\textsuperscript{7.3} that PoX schemes can be used to construct sensors and actuators that “cannot 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 \[15,28\]. However, such architectures are comparatively heavy-weight and unsuitable for low-end devices.

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 ($\mathcal{V}$) to remotely measure memory contents (or software state) of an untrusted embedded device ($\mathcal{P}_{\text{rv}}$). RA is usually realized as a 2-message challenge-response protocol:
1. $\Psi_{rf}$ sends an attestation request containing a challenge ($Chal$) to $Prv$. It might also contain a token derived from a secret (shared by $\Psi_{rf}$ and $Prv$) that allows $Prv$ to authenticate $\Psi_{rf}$.
2. $Prv$ receives the attestation request, authenticates the token (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 $\Psi_{rf}$.
4. $\Psi_{rf}$ receives the result, and decides whether it corresponds to a valid memory state.

The authenticated integrity check is typically realized as a Message Authentication Code (MAC) computed over $Prv$ memory. We discuss one concrete RA architecture for $Prv$ in Section 2.

Despite major progress and several proposals for RA architectures with different assumptions and guarantees [6, 8, 10, 11, 17, 21, 22, 27, 32, 36, 39, 42], RA alone is insufficient to obtain proofs of execution. RA allows $\Psi_{rf}$ to ascertain integrity of software residing in $Prv$ attested memory region. However, RA by itself 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 elsewhere in memory) in 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 VAPE: Verified Architecture for Proofs of Execution. In addition to RA, VAPE allows $\Psi_{rf}$ to request an unforgeable proof that the attested software executed successfully and (optionally) produced certain authenticated output. These guarantees hold even in case of full software compromise on $Prv$. Our intended contributions are:

- **New security service:** we design and implement VAPE for unforgeable remote proofs of execution (PoX). VAPE is composed with VRASED [17], a formally verified hybrid RA architecture. As we discuss in the rest of this paper, obtaining provably secure PoX requires significant architectural support in addition to a secure RA functionality (see Section 7). Nonetheless, we show that VAPE careful design achieves all necessary properties for secure PoX at 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 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 requirements, leading to the incorrect conclusion that PoX can be obtained directly from RA; see Section 2 for examples. In this work, we prove that VAPE yields a secure PoX architecture. All security properties expected from VAPE implementation are formally specified using Linear Temporal Logic (LTL) and VAPE modules are verified to adhere to these properties. We also prove that the composition of VAPE new modules with a formally verified RA architecture (VRASED) implies a concrete definition of PoX security.

- **Evaluation, publicly available implementation and applications:** VAPE was implemented on a real-world low-end MCU (TI MSP430) and deployed using commodity FPGAs. Both design and verification are publicly available at [2]. 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 VAPE to construct a trustworthy safety-critical device, on which malware can not spoof execution results (e.g., spoof sensed values) without detection.

**Targeted Devices & Scope:** This work focuses on CPS/IoT sensors and actuators with relatively low computing power. These 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 $\approx 64$ KBytes of addressable memory. SRAM is used as data memory and its size is normally within 4-16KBytes 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 [25]. 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.

**Organization:** Section 2 discusses related work on remote attestation, formal verification of security services and control flow attestation. Section 3 provides some background on automated verification, and VRASEDRA architecture. Section 4 introduces Proofs of Execution (PoX), followed by a realization thereof in Section 5 including technical details of VAPE design, as well as the adversarial model and assumptions. Section 6 presents VAPE’s formal verification. Next, in Section 7 we report VAPE’s evaluation results and describe how to use VAPE to implement authenticated sensing/actuation. Section 8 concludes the paper with a summary of results.

2 Related Work

Remote Attestation (RA)– architectures fall into three categories: hardware-based, software-based, or hybrid. Hardware-based [34, 40, 46] relies on dedicated secure hardware components, e.g., Trusted Platform Modules (TPMs) [45]. However,
the cost of such hardware is normally prohibitive for low-end IoT/CPS devices. Software-based attestation \cite{30,43,44} requires no hardware security features but imposes strong security assumptions about communication between \(P_{rv}\) and \(\forall rf\), which are unrealistic in the IoT/CPS ecosystem (though, it is the only choice for legacy devices). Hybrid RA \cite{7,21,22,24,33} aims to achieve security equivalent to hardware-based mechanisms at minimal cost. It thus entails minimal hardware requirements while relying on software to reduce overall complexity and RA footprint on \(P_{rv}\).

The first hybrid RA architecture -- SMART \cite{22} -- acknowledged the importance of proving remote code execution on \(P_{rv}\) in addition to just attesting \(P_{rv}\)'s memory. Using an \textit{attest-then-execute} approach (see Algorithm 4 in \cite{22}), SMART attempts to achieve software execution guarantees by specifying the address of the first instruction to be executed after completion of attestation. We consider this to be a best-effort approach which merely guarantees that the code \textit{starts executing}. However, it \textit{does not guarantee that execution completes successfully}. For example, SMART’s approach cannot detect if execution is interrupted (e.g., by malware) and never resumed. It also cannot detect when a reset (e.g., due to software bugs, or \(P_{rv}\) running low on power) happens during execution, thus preventing its successful completion. Furthermore, direct memory access (DMA) can occur during execution and it can modify the code being executed, or its output. In other words, SMART offers no guarantees beyond “invoking the executable”.

Another notable RA architecture is TrustLite \cite{32}, which builds upon SMART to allow secure interrupts. However, TrustLite does not enforce temporal consistency of attested memory; it is thus conceptually vulnerable to self-relocating malware and memory modification during attestation \cite{9}. Consequently, it is challenging to derive secure PoX from TrustLite. Several other prominent low-to-medium-end RA architectures -- e.g., SANCUS \cite{37}, HYDRA \cite{21}, and TyTaN \cite{7} -- do not offer PoX. In this paper, we show that the \textit{execute-then-attest} approach, using a temporally consistent RA architecture, provides unforgeable proofs of execution that are produced only if the execution and its results are not tampered with, and it completes successfully.

\textbf{Control Flow Attestation (CFA)} -- In contrast with RA, which measures \(P_{rv}\)'s software integrity, CFA techniques \cite{1,18,19,47} provide \(\forall rf\) with a measurement of the exact control flow path taken during execution of specific software on \(P_{rv}\). Such measurements allow \(\forall rf\) 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 \cite{1} assumes ARM TrustZone, while LO-FAT \cite{19} and LiteHAX \cite{18} 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) \cite{45}, is about 10× more expensive than MSP430 MCU itself.

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 \(\forall rf\) 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 4.2 for a discussion on run-time threats).

Instead of relying on CFA, our work constructs a PoX-specific architecture -- VAPE -- that enables low-cost PoX for low-end devices. VAPE 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, \(\forall rf\) 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.

\textbf{Formally Verified Security Services} -- In recent years, several efforts focused on formally verifying security-critical systems. In terms of cryptographic primitives, Hawblitzel et al. \cite{26} verified implementations of SHA, HMAC, and RSA. Bond et al. \cite{5} verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. Zinzindohoué, et al. \cite{48} developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl \cite{3}. Larger security-critical systems have also been successfully verified. Bhargavan \cite{4} implemented the TLS protocol with verified cryptographic security. CompCert \cite{35} is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. \cite{31} designed and proved functional correctness of the seL4 microkernel. More recently, VRASED \cite{17} realized a formally verified hybrid RA architecture. VAPE architecture, proposed in this paper, uses VRASED RA functionality (see Section 5.2 for details) composed with additional formally verified architectural components to obtain provably secure PoX.

\section{Background}

\subsection{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, \footnote{Source: https://www.digikey.com/}
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 model checker NuSMV [13], 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 [29], 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:

- \(\Box \phi\) – \(\neg \Delta \phi\): holds if \(\phi\) is true at the next system state.
- \(\Diamond \phi\) – Future \(\phi\): holds if there exists a future state where \(\phi\) is true.
- \(\Box \phi\) – Globally \(\phi\): holds if for all future states \(\phi\) is true.
- \(\phi \Upsilon \psi\) – Until \(\psi\): holds if there is a future state where \(\psi\) holds and \(\phi\) holds for all states prior to that.
- \(\phi \Gamma \psi\) – 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: \(\phi \Gamma \psi \equiv \neg (\neg \phi \Upsilon \psi)\).

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

\textit{VRASED} [17] 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 \textit{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 \textit{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, \textit{VRASED} software (Sw-Att) is stored in Read-Only Memory (ROM) and relies on a (previously) formally verified HMAC implementation from HACL* cryptographic library [48]. 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 \(K\).
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.

\textit{VRASED} hardware (Hw-Mod) monitors 7 MCU signals:

- \(PC\): Current Program Counter value;
- \(R_{en}\): Signal that indicates if the MCU is reading from memory (1-bit);
- \(W_{en}\): Signal that indicates if the MCU is writing to memory (1-bit);
- \(D_{addr}\): Address for an MCU memory access;
- \(DMA_{en}\): 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 \textit{VRASED}’s hardware detects any violation of its security properties. \textit{VRASED} hardware is described in Register Transfer Level (RTL) using Finite State Machines (FSMs). Then, NuSMV Model Checker [14] 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, \textit{VRASED} end-to-end security proof guarantees that no probabilistic polynomial time (PPT) adversary can win the RA security game with non-negligible probability in terms of the security parameter. (See Definition in Appendix [3]).
Definition 1 (Proof of Execution (PoX) Scheme).
A Proof of Execution (PoX) scheme is a tuple of algorithms \([XRequest, XAtomicExec, XProve, XVerify]\) performed between \(\mathcal{P}_r\) and \(\mathcal{V}_r\) where:

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

2. \(XAtomicExec^{\mathcal{P}_r}(ER_r):\) an algorithm (with possible hardware-support) that takes as input some executable region \(ER_r\) in \(\mathcal{P}_r\)'s memory; containing a list of instructions \(\{i_1, i_2, ..., i_n\}\). \(XAtomicExec\) runs on \(\mathcal{P}_r\) and is considered successful if: (1) instructions in \(ER_r\) are executed from its first instruction, \(i_1\), and end at its last instruction, \(i_n\); (2) \(ER_r\)’s execution is atomic, i.e., if \(E\) is the sequence of instructions executed between \(i_1\) and \(i_n\), then \(\{e\} \subseteq E\); and (3) \(ER_r\)’s execution flow is not altered by external events, i.e., MCU interrupts or DMA events. The \(XAtomicExec\) algorithm outputs a string \(O\). Note that \(O\) may be a default string \(\$\) if \(ER_r\)’s execution does not result in any output.

3. \(XProve^{\mathcal{P}_r}(ER_r, \mathcal{Ch}_a, O_r):\) an algorithm (with possible hardware-support) that takes as input some \(ER_r\), \(\mathcal{Ch}_a\) and \(O_r\) and is run by \(\mathcal{P}_r\) to output \(H\), i.e., a proof that \(XRequest^{\mathcal{V}_r}S_r\) and \(XAtomicExec^{\mathcal{P}_r}(ER_r)\) happened (in this sequence) and that \(O_r\) was produced by \(XAtomicExec^{\mathcal{P}_r}(ER_r)\).

4. \(XVerify^{\mathcal{V}_r}(H_r, \mathcal{Ch}_a, O_r):\) an algorithm executed by \(\mathcal{V}_r\) with the following inputs: some \(\mathcal{Ch}_a\), \(H_r\) and \(O_r\). The \(XVerify\) algorithm checks whether \(H_r\) is a valid proof of the execution of \(S_r\) (i.e., executed memory region \(ER_r\) corresponds to \(S_r\)) on \(\mathcal{P}_r\) given the challenge \(\mathcal{Ch}_a\), and if \(O_r\) is an authentic output/result of such an execution. If both checks succeed, \(XVerify\) 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 \(\mathcal{V}_r\) issues \(\mathcal{Ch}_a \leftarrow XRequest^{\mathcal{V}_r}S_r\).
- Let \(t_{verif}\) denote time when \(\mathcal{V}_r\) receives \(H_r\) and \(O_r\) back from \(\mathcal{P}_r\) in response to \(XRequest^{\mathcal{V}_r}\).
- Let \(XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\) denote that \(XAtomicExec^{\mathcal{P}_r}(ER_r, \cdot)\), such that \(ER_r = S_r\), was invoked and completed within the time interval \([t_{req}, t_{verif}]\).
- Let \(O = XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\) denote that \(XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\) produces output \(O_r\). Conversely, \(O \neq XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\) indicates \(O_r\) is not produced by \(XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\).

\(\mathcal{P}_r\) PoX Security Game (PoX-game): Challenger plays the following game with \(Adv\):
1. \(Adv\) is given full control over \(\mathcal{P}_r\) software state and oracle access to calls to the algorithms \(XAtomicExec^{\mathcal{P}_r}\) and \(XProve^{\mathcal{P}_r}\).
2. At time \(t_{req}\), \(Adv\) is presented with software \(S_r\) and challenge \(\mathcal{Ch}_a\).
3. \(Adv\) wins in two cases:
   (a) None or incomplete execution: \(Adv\) produces \((H_{adv}, O_{adv})\), such that \(XVerify(H_{adv}, O_{adv}, S_r, \mathcal{Ch}_a) = 1\), without calling \(XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\).
   (b) Execution with tampered output: \(Adv\) calls \(XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\) and can produce \((H_{adv}, O_{adv})\), such that \(XVerify(H_{adv}, O_{adv}, S_r, \mathcal{Ch}_a) = 1\) and \(O_{adv} \neq XAtomicExec^{\mathcal{P}_r}(S_r, t_{req} \rightarrow t_{verif})\).

\(\mathcal{P}_r\) PoX Security Definition:
A PoX scheme is considered secure for security parameter \(1\) if, for all PPT adversaries \(Adv\), there exists a negligible function \(\text{negl}(1)\) such that:

\[
Pr[Adv, \text{PoX-game} \leq \text{negl}(1)]
\]

4 Proof of Execution (PoX) Schemes

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

We now justify the need to include atomic execution of \(S_r\) 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 \(\mathcal{P}_r\) 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 Adv that might control Prv’s entire software state, code, and data. Adv can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules. Adv may also have full control over all Direct Memory Access (DMA) controllers of Prv. 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, XProve, XVerify) to be secure if the aforementioned Adv has only negligible probability of convincing \( \psi/\varphi \) 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, Adv cannot 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 \( \psi/\varphi \) issuing the request and receiving the response. Therefore, if a PoX scheme is secure according to this definition, \( \psi/\varphi \) 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 offer a guarantee that S, as specified by \( \psi/\varphi \), 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 Prv. Since \( \psi/\varphi \) 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 12 or static analysis 15) 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 S to Prv and the whole issue is orthogonal to PoX functionality.

Physical Attacks: physical and hardware-focused attacks are out of scope of this paper. Specifically, we assume that Adv cannot modify code in ROM, induce hardware faults, or retrieve Prv secrets via physical presence side-channels. Protection against such attacks is considered orthogonal and could be supported via standard physical security techniques 41.

4.2 MCU Assumptions

VAPE is composed with VRASED to enable a verified architecture for proofs of execution. Therefore, 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.

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

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

**A4 – MCU Reset:** At the end of a successful reset routine, all registers (including PC) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine cannot 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 irq signal is set.

5 VAPE: A Secure PoX Architecture

We now present VAPE, a new PoX architecture that realizes the PoX security definition in Definition 2. The key aspect of VAPE is a computer-aided formally verified and publicly available implementation thereof. This section first provides some intuition behind VAPE’s design. All VAPE 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 software” to refer to any software other than SW-Att code from VRASED. Adv is allowed to overwrite or bypass any “unprivileged software”. Meanwhile, “trusted software” refers to VRASED’s implementation of SW-Att (see Section 3) which is formally verified and can not be modified by Adv, since it is stored in ROM. VAPE is designed such that no changes to
Definition 3 (Proof of Execution Protocol). VAPE instantiates a PoX = (XRequest, XAtomicExec, XProve, XVerify) scheme behaving as follows:

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

2. XAtomicExec$^{\Psi_{\mathit{RE}}}$ (ER, OR, METADATA): This algorithm starts with unprivileged auxiliary software writing the values of: ER$_{\text{min}}$, ER$_{\text{max}}$, OR$_{\text{min}}$, OR$_{\text{max}}$ and Chal to a special pre-defined memory region denoted by METADATA. VAPE’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 S by setting the program counter to the value of ER$_{\text{min}}$.

3. XProve$^{\Psi_{\mathit{RE}}}$ (E, Chal, OR): produces proof of execution $\mathcal{H}$. $\mathcal{H}$ allows $\Psi_{\mathit{RE}}$ to decide whether: (1) code contained in ER actually executed; (2) ER contained specified (expected) 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, VAPE 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:

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

METADATA also contains the EXEC flag that is read-only to all software running in PoX and can only be written to by VAPE’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 PoX attempts to violate any of these properties, VAPE’s verified hardware (provably) sets EXEC to zero. After computing $\mathcal{H}$, PoX returns it and contents of OR (O) produced by ER’s execution to $\Psi_{\mathit{RE}}$.

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

\[
\mathcal{H} = \text{HMAC}(\text{KDF}(\chi, \text{Chal}^{\Psi_{\mathit{RE}}}), S, O, \text{METADATA}^{\Psi_{\mathit{RE}}},...) \tag{2}
\]

Table 1: Notation

| Symbol  | Description                                      |
|---------|--------------------------------------------------|
| PC      | Current Program Counter value                    |
| R$_{\text{do}}$ | Signal that indicates if the MCU is reading from memory (1-bit) |
| W$_{\text{do}}$ | Signal that indicates if the MCU is writing to memory (1-bit) |
| DMA$_{\text{do}}$ | Address for an MCU memory access |
| DMA$_{\text{abled}}$ | Memory address being accessed by DMA, if any |
| sig     | Signal that indicates if an interrupt is happening |
| CR      | Memory region where SW-Att is stored: CR = [CR$_{\text{min}}$, CR$_{\text{max}}$] |
| MR      | (MAC Region) Memory region in which SW-Att computation result is written: MR = [MR$_{\text{min}}$, MR$_{\text{max}}$]. The same region is used to pass the attestation challenge as input to SW-Att |
| AR      | (Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\Psi_{\mathit{RE}}$. AR = [AR$_{\text{min}}$, AR$_{\text{max}}$] |
| XS      | (Exclusive Stack Region) Exclusive memory region that contains SW-Att’s stack and can be only accessed by SW-Att |
| reset   | A 1-bit signal that reboots/resetes the MCU when set to logical 1 |
| ER      | (Execution Region) Memory region that stores an executable to be executed: ER = [ER$_{\text{min}}$, ER$_{\text{max}}$] |
| OR      | (Output Region) Memory region that stores execution output: OR = [OR$_{\text{min}}$, OR$_{\text{max}}$] |
| EXEC    | 1-bit execution flag indicating whether a successful execution has happened |
| METADATA | Memory region containing VAPE’s metadata |

Sw-Att are required. Therefore, both functionalities (RA and PoX, i.e., VRASED and VAPE) can co-exist on the same device without interfering with each other.

Notation is summarized in Table 1.

5.1 Protocol and Architecture

VAPE implements a secure PoX = (XRequest, XAtomicExec, XProve, XVerify) scheme conforming to Definition 3. The steps in VAPE workflow are illustrated in Figure 1. The main idea is to first execute code contained in ER. Then, at some later time, VAPE invokes VRASED verified RA functionality to attest the code in ER and include,
in the attestation result, additional information that allows \( Vrf \) to verify that \( ER \) code actually executed. If \( ER \) execution produces an output (e.g., \( Prv \) 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 \( VAPE \) formally verified hardware and its memory cannot be written by any software running on \( Prv \). \( VAPE \) hardware module runs in parallel with the MCU, monitoring its behavior and deciding the value of \( EXEC \) accordingly.

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

\( METADATA \) includes the \( EXEC \) flag, which is initialized to 0 and only changes from 0 to 1 (by \( VAPE \)'s hardware) when \( ER \) execution starts, i.e., when the PC points to \( ER_{\text{min}} \). Afterwards, any violation of \( VAPE \)'s security properties (detailed in Section 3.2) 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=\( ER_{\text{min}} \). In other words, \( VAPE \) verified hardware makes sure that \( EXEC \) value covered by the HMAC’s result (represented by \( Vrf \)) 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 \( ER_{\text{min}} \) to its last instruction \( ER_{\text{max}} \).

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

**Remark:** Our notion of successful execution requires \( S \) to have a single exit point – \( ER_{\text{max}} \). 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 \( ER_{\text{max}} \). 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 Appendix C.

Finally, \( Vrf \) is responsible for defining \( OR \) memory region according to \( S \) behavior. \( OR \) should be large enough to fit all output produced by \( S \) and \( OR \) boundaries should correspond to addresses where \( S \) writes its output values to be sent to \( Vrf \).

### 5.2 \( VAPE \)'s Sub-Properties at a High-Level

We now describe sub-properties enforced by \( VAPE \). Section 6 formalizes them in LTL and provides a single end-to-end definition of \( VAPE \) correctness. This end-to-end correctness notion is provably implied by the composition of all sub-properties. Sub-properties fall into two major groups: **Execution Protection** and **Metadata Protection**. A violation of any of these properties implies one or more of:

- Code in \( ER \) was not executed atomically and in its entirety;
- Output in \( OR \) was not produced by \( ER \) execution;
- Code in \( ER \) was not executed in a timely manner, i.e., after receiving the latest \( XRequest \).

Whenever \( VAPE \) detects a violation, \( EXEC \) is set to 0. Since \( EXEC \) is included among inputs to the computation of HMAC (conveyed in \( Prv \)'s response), it will be interpreted by \( Vrf \) as failure to prove execution of code in \( ER \).

**Remark:** We emphasize that properties discussed below are required in addition to \( VRASED \) verified properties, i.e., these are entirely different properties used specifically to enforce PoX security and should not be viewed as replacements for any of \( VRASED \) properties that are used to enforce RA security.

#### 5.2.1 Execution Protection:

**EP1 – Ephemeral Immutability:** Code in \( ER \) cannot be modified from the start of its execution until the end of \( Sw\)\(\text{-At}\)\(\text{t} \) computation in \( XProve \) routine. This property is necessary to ensure that the attestation result reflects the code that executed. Lack of this property would allow \( Adv \) to execute some other code \( ER_{\text{Adv}} \), overwrite it with expected \( ER \) and finally call \( XProve \). This would result in a valid proof of execution of \( ER \) even though \( ER_{\text{Adv}} \) was executed instead.
EP2 – Ephemeral Atomicity: ER execution is only considered successful if ER runs starting from ERmin until ERmax atomically, i.e., without any interruption. This property conforms with XAtomicExec routine in Definition 4 and with the notion of successful execution in the context of our work. As discussed in Section 4, ER must run atomically to prevent malware residing on Tprov from interrupting ER execution and resuming it at a different instruction, or modifying intermediate execution results in data memory. Without this property, Return-Oriented Programming (ROP) and similar attacks on ER could change its behavior completely and unpredictably, making any proof of execution (and corresponding output) useless.

EP3 – Output Protection: Similar to EP1, VAPE must ensure that OR is unmodified from the time after ER code execution is finished until completion of HMAC computation in XProve. Lack of this property would allow Adv to overwrite OR and successfully spoof OR produced by ER, thus convincing \( \psi r_f \) that it produced output ORAdv.

5.2.2 Metadata Protection:

MP1 - Executable/Output (ER/OR) Boundaries: VAPE hardware ensures properties EP1, EP2, and EP3 according to values: ERmin, ERmax, ORmin, ORmax. These values are configurable and can be decided by \( \psi r_f \) based on application needs. They are written into metadata-dedicated physical addresses in Tprov memory before ER execution. Therefore, once ER execution starts, VAPE hardware must ensure that such values remain unchanged until XProve completes. Otherwise, Adv could generate valid attestation results, by attesting [ERmin, ERmax], while, in fact, having executed code in a different region: [ERAdv, ERAdv].

MP2 - Response Protection: The appropriate response to \( \psi r_f \)’s challenge must be unforgeable and non-invertible. Therefore, in the XProve routine, \( \kappa \) 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, VAPE guarantees that no software in Tprov can ever modify EXEC flag and that, once EXEC = 0, it can only become 1 if ER’s execution re-starts afresh.

MP3 - Challenge Temporal Consistency: VAPE 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) Tprov-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 \( \psi r_f \) 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 VAPE sub-properties Linear Temporal Logic (LTL) to define invariants that must hold throughout the MCU operation. We then use a theorem prover [20] 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 VAPE hardware. VAPE 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 VAPE security.

VAPE 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 [29]. The resulting SMV is simultaneously verified against all LTL specifications, using the model checker NuSMV [12], to prove that the final Verilog of VAPE 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 VAPE’s implementation.

Definition 4. Machine Model (subset)
1. Modify_Mem(i) \( \rightarrow \) (We\(n\) \& \( D_{addr} = i \)) \( \lor \) (DMA\(m\) \& \( DMA_{addr} = i \))
2. Interrupt \( \rightarrow \) irq
3. MR, CR, AR, KR, XS, and METADATA are non-overlapping memory regions

Modify_Mem models that a given memory address can be modified by a CPU instruction or by a DMA access. In the former, We\(n\) signal must be set and D_{addr} must contain the target memory address. In the latter, DMA\(m\) signal must be set and DMA_{addr} must contain the target DMA address. The requirements for reading from a memory address are similar, except that instead of We\(n\), R\(e\)n must be on. We do not explicitly state this behavior since it is not used in VAPE 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 VAPE to store information about software execution status.

6.2 Security & Implementation Correctness

We use a two-part strategy to prove that VAPE 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 t prv. To show this, we compose a computer proof using SPOT LTL proof assistant [20].

[B]: We use cryptographic reduction proofs to show that, as long as part A holds, VRASED security can be reduced to VAPE’s PoX security from Definition 2. In turn, HMAC’s existential unforgeability can be reduced to VRASED’s security [17]. Therefore, both VAPE 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 VAPE’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 ∧ PC ∈ CR] to hold), at least once before such time the following must have happened:

1. The system reached state S0 where software stored in ER started executing from its first instruction ($PC = ER_{min}$).
2. The system eventually reached a state S1 when ER finished executing ($PC = ER_{max}$). In the interval between S0 and S1, PC kept executing instructions within ER, there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S2 when XProve started executing ($PC = CR_{min}$). In the interval between S0 and S2, METADATA and ER regions were not modified.
4. In the interval between S0 and S2, OR region was only modified by ER’s execution, i.e., $PC ∈ ER ∨ ¬ Modify _ Mem (OR)$.

Figure 2 shows the time windows wherein each memory region must not change during VAPE’s PoX as implied by VAPE’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 $∀ rf$ rejecting the proof of execution since it will not conform to the expected value of $H$, per Equation 2 in Definition 3.

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

6.3 VAPE’s Sub-Properties in LTL

We formalize the necessary sub-properties enforced by VAPE as LTL specifications [3, 12] in Definition 2. 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 VAPE 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 – Ephemerality 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, 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 ensuring that ER always executes in its entirety. Finally, LTL 6 enforces that EXEC is set to zero if a interrupt happens in the middle of ER execution. Even though LTLs 4 and 5 already enforce that PC cannot 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 can only modify OR when executing instructions within ER; and 3) DMA cannot 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 Chal as well as $ER_{min}$, $ER_{max}$, OR$_{min}$, and OR$_{max}$ are all stored in METADATA reserved memory region, it suffices to ensure that EXEC is set to logical 0 whenever this region is modified. Also, LTL 8 enforces that EXEC is only set to one if ER and OR are configured (by METADATA values $ER_{min}$, $ER_{max}$, OR$_{min}$, OR$_{max}$) as valid memory regions.
Finally, LTLs \(11\) and \(12\) (in addition to VRASED verified RA architecture) are responsible for ensuring MP2-Response Protection by making sure that EXEC always reflects what is intended by VAPE hardware. LTL \(7\) specifies that the only way to change 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 EXEC is 0.

To conclude, recall that EXEC is read-only to all software running on \(P_{Trv}\). Therefore, malware can not change it directly.

VAPE is designed as a set of seven hardware sub-modules, each verified to enforce a subset of properties discussed in this section. Due to space constraints, examples of implementation of verified sub-modules as FSMs are discussed in Appendix A.

### 7 Implementation & Evaluation

VAPE implementation uses OpenMSP430 \(25\) as its open core implementation. We implement the hardware architecture shown in Figure 2. In addition to VAPE and VRASED modules in HW-Mod, we implement a peripheral module responsible for storing and maintaining VAPE 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 un-modifiable in software by removing software-write wires in hardware. Finally, we use Xilinx Vivado to synthesize an RTL description of the modified HW-Mod and deploy it on the Artix-7 FPGA class.
7.1 Evaluation Results

Hardware & Memory Overhead. Table 2 reports VAPE hardware overhead as compared to unmodified OpenMSP430 [25] and VRASED [17]. VAPE 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, VAPE 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 VAPE-enabled PoX since VAPE does not introduce new instructions or modifications to the MSP430 ISA. VAPE 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-An is a sequence of instructions or have long loops). As mentioned earlier, VAPE does not affect \( S \) runtime. XProve’s run-time is linear in the size of \( ER + OR \). In the worst-case scenario where these regions occupy the entire program 8kB memory, XProve takes around 900ms on an 8MHz device.

Verification Efforts. We verify VAPE on an Ubuntu 16.04 machine running at 3.40GHz. Results are shown in Table 2. VAPE verification requires checking 10 extra invariants (shown in Definition 3) 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, VAPE 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 VAPE overhead with that state-of-the-art CFA architectures. As discussed in Section 7 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 [47], LiteHAX [18], and LO-FAT [19]. Figure 4.a compares VAPE 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 (VAPE’s dedicated RAM corresponds to the exclusive access stack implemented by VRASED).

As expected, VAPE 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-end devices. Meanwhile, VAPE needs a total of 12% additional LUTs and 2% additional registers. VRASED requires about 2 kB of reserved RAM, which is not increased by VAPE PoX support.

7.3 Proof of Concept: Authenticated Sensing and Actuation

As discussed in Section 7 an important functionality attainable with PoX is authenticated sensing/actuation. In this section, we demonstrate how to use VAPE to build sensors and actuators that “cannot 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 \( CO_2 \) 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 autho-
Hardware Reserved Verification

|           | Hardware | Reserved | # LTL Invariants | Verification | Time (s) | Mem (MB) |
|-----------|----------|----------|------------------|--------------|----------|----------|
|           | Reg LUT  | RAM (byte) |                  |              |          |          |
| OpenMSP430 | 691     | 1904     | 0                |              |          |          |
| VRASED    | 721     | 1964     | 2332             | 10           | 0.4      | 13.6     |
| VAPE + VRASED | 735 | 2206     | 2341             | 20           | 1385     | 183.6    | 280.3    |

Table 2: Evaluation results.

Figure 4: Overhead comparison between VAPE and CFA architectures. Dashed lines in (a) and (b) represent the total hardware cost of MSP430. Dashed line in (c) represents total addressable memory (64 kB) on MSP430.

As a proof of concept, we use VAPE to implement a simple fire sensor that operates with temperature and humidity quantities. It communicates with a remote Vr (e.g., the fire department) using a low-power ZigBee radio (typically used by low-end CPS/IoT devices). Temperature and humidity analog devices are connected to a VAPE-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.

VAPE is used to prove execution of the fire sensor software. This software is shown in Figure 8 in Appendix D. 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 and was ported to a VAPE-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 VAPE’s PoX architecture. Details can be found in Appendix D.

As a proof of concept, we use VAPE to implement a simple fire sensor that operates with temperature and humidity quantities. It communicates with a remote Vr (e.g., the fire department) using a low-power ZigBee radio (typically used by low-end CPS/IoT devices). Temperature and humidity analog devices are connected to a VAPE-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.

VAPE is used to prove execution of the fire sensor software. This software is shown in Figure 8 in Appendix D. 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 and was ported to a VAPE-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 VAPE’s PoX architecture. Details can be found in Appendix D.

3https://www.zigbee.org/
4https://github.com/Seeed-Studio/LaunchPad_Kit
8 Conclusion

This paper introduces VAPE, 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 VAPE’s use in many IoT application domains, such as authenticated sensing and actuation. Our implementation of VAPE is realized on a real embedded system platform, MSP430, synthesized on an FPGA, and the verified implementation is publicly available. Our evaluation shows that VAPE has low overhead for both hardware footprint and time for generating proofs of execution.

References

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

[2] Anonymous Authors. VAPE source code. https://www.dropbox.com/sh/9id1ntfrnjy40tc/AADONZgUdibXlONxSMdlm6pa, 2018.

[3] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In International Conference on Cryptology and Information Security in Latin America, 2012.

[4] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In SP, 2013.

[5] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In USENIX, 2017.

[6] Ferdinand Brasser, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote attestation for low-end embedded devices: the prover’s perspective. In DAC, 2016.

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

[8] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In ASIACCS, 2018.

[9] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security, pages 313–327. ACM, 2018.

[10] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In Design, Automation and Test in Europe (DATE), 2018.

[11] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. Remote attestation of iot devices via SMARM: Shuffled measurements against roving malware. In IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2018.

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

[13] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In International Conference on Computer Aided Verification, pages 359–364. Springer, 2002.

[14] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In International Conference on Computer Aided Verification, pages 359–364. Springer, 2002.

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanc-tum: Minimal hardware extensions for strong software isolation. In 25th {USENIX} Security Symposium ({USENIX} Security 16), 2016.

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

[17] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. Vrased: A verified hardware/software co-design for remote attestation. USENIX Security’19 (To appear). Pre-print available at: https://arxiv.org/abs/1811.00175 2019.

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

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

[20] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In International Symposium on Automated Technology for Verification and Analysis, pages 122–129. Springer, 2016.

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

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

[23] Karim Eldefrawy et al. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In NDSS, 2012.

[24] Aurélien Francillon et al. A minimalist approach to remote attestation. In DATE, 2014.

[25] Olivier Girard. openMSP430, 2009.

[26] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In OSDI, volume 14, pages 165–181, 2014.

[27] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SeED: secure non-interactive attestation for embedded devices. In ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), 2017.

[28] Intel. Intel Software Guard Extensions (Intel SGX).

[29] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A tool for word-level verification. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016, pages 1156–1159. IEEE, 2016.

[30] Rick Kennell et al. Establishing the genuinity of remote computer systems. In USENIX, 2003.

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elka-duwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP ’09, pages 207–220, New York, NY, USA, 2009. ACM.

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

[33] P. Koebler et al. TrustLite: A security architecture for tiny embedded devices. In EuroSys, 2014.

[34] X. Ková et al. New results for timing-based attestation. In IEEE S&P ’12, 2012.

[35] Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107–115, 2009.

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

[37] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for iot devices. ACM Trans. Priv. Secur., 20(3):7:1–7:33, July 2017.

[38] Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Towards systematic design of collective remote attestation protocols. In ICDCS, 2019.

[39] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In ESORICS, 2010.

[40] Jr. Petroni et al. Copilot — A coprocessor-based kernel runtime integrity monitor. In USENIX, 2004.

[41] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In VLSI Design, 2004. Proceedings. 17th International Conference on, pages 605–611. IEEE, 2004.

[42] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In ACM workshop on Wireless security, 2006.

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

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

[45] Trusted Computing Group. Trusted platform module (tpm), 2017.
VAPE is designed as a set of seven sub-modules. We now describe VAPE’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 [2]. Each sub-module enforces a sub-set of the LTL specifications in Definition 4. 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 signals shown in Figure 2 and produces only one output — EXEC — indicating violation 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 VAPE is the conjunction (logical AND) of all sub-modules’ individual EXEC flags.

Figure 6 represents a verified model enforcing LTLs 4–6, corresponding to the high-level property EP2- Epphemeral 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 (ER_min) and last (ER_max) instructions. Meanwhile, lstER and fstER correspond to states when PC points to the first and last instructions, respectively. The only possible path from notER to midER is through fstER. 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.

APPENDIX

A Sub-Module Verification

VAPE is designed as a set of seven sub-modules. We now describe VAPE’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 [2]. Each sub-module enforces a sub-set of the LTL specifications in Definition 4. 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 signals shown in Figure 2 and produces only one output — EXEC — indicating violation 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 VAPE is the conjunction (logical AND) of all sub-modules’ individual EXEC flags.

Figure 6 represents a verified model enforcing LTLs 4–6, corresponding to the high-level property EP2- Epphemeral 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 (ER_min) and last (ER_max) instructions. Meanwhile, lstER and fstER correspond to states when PC points to the first and last instructions, respectively. The only possible path from notER to midER is through fstER. 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 for Implementation Correctness & Security

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

---

**Theorem 1.** Definition 4 ∧ LTLs 3 –12 → Definition 5
proof for Theorem 1 is available at [2]. We here discuss the intuition behind such proof. Theorem 1 states that LTLs [1]–[2] when considered in conjunction with the machine model in Definition 3 imply VAPE’s implementation correctness.

Recall that Definition 3 states that, in order to have EXEC = 1 during the computation of XProve, at least once before such time 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 = ER_min).

2. The system eventually reached a state S₁ when ER finished executing (PC = ER_max). In the interval between S₀ and S₁, PC remained executing instructions within ER, there were no interrupts, no resets, and DMA remained inactive.

3. The system eventually reached a state S₂ when XProve started executing (PC = CR_min). In the interval between S₀ and S₂ the memory regions of METADATA and ER were not modified.

4. In the interval between S₀ and S₂ the OR memory region was only modified by ER’s software execution (PC ∈ ER ∨ ¬ Modify_Mem(OR)).

The first two properties to be noted are LTL [2] and LTL [1]. LTL [2] establishes the default state of EXEC is 0. LTL [1] enforces that the only possible way to change EXEC from 0 to 1 is by having PC = ER_min. In other words, EXEC is 1 during the computation of XProve only if, at some before that, the code stored in ER started to execute (state S₀).

To see why state S₁ (when ER execution finishes, i.e., PC = ER_max) is reached and until then ER executes atomically, we look at LTLs [4] and [6] and LTLs [1] and [3] and [8] ensure that PC will stay inside ER until S₁ or otherwise EXEC will set to 0. On the other hand, it is impossible to execute instructions of XProve (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 guarantees that ER and CR do not overlap, or EXEC = 0.

Clearly, the contents of ER cannot 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 2 is strictly implied by VAPE’s implementation. This concludes the reasoning behind Theorem 1.

Theorem 2. VAPE is secure according to Definition 2 as long as VRASED is a secure RA architecture according to Definition 4.

Definition 7. VRASED’s Security Game [17]

1. **RA Security Game (RA-game):**
   - Let t be the probability that an adversary wins.
   - Let d be the probability that an adversary wins.
   - Let l be the probability that an adversary wins.
   - Let M be the probability that an adversary wins.

   **Notation:**
   - I is the security parameter and |I| = |Chal| = |MR| = l
   - AR(t) denotes the content of AR at time t.
   - RA denotes the RA architecture.
   - SW-Att denotes the security game against SW-Att.
   - XAtomicExec denotes the security game against XAtomicExec.
   - XVerify denotes the security game against XVerify.
   - HMAC denotes the security game against HMAC.

   **RA Security Definition:**
   - An RA scheme is considered secure if for all PPT adversaries Adv, there exists a negligible function negl such that:
     
     \[ Pr[Adv, RA-game] = \leq negl(l) \]

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

First we recall that to win VAPE’s security game Adv_ROM must provide \( \{H_{Adv}, O_{Adv}\} \), such that XVerify\( (H_{Adv}, O_{Adv}, S, Chal, \cdot) = 1 \). To comply with conditions 3.a and 3.b in Definition 2, this must be done either of the following two cases:

**Case 1.** Adv does not execute S in the time window between t_req and t_verif (i.e., \( \neg XAtomicExec \rightarrow t_{req} \rightarrow t_{verif} \)).

**Case 2.** Adv calls XAtomicExec \( \rightarrow t_{req} \rightarrow t_{verif} \) 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 VAPE’s XVerify algorithm (see Definition 3) a token \( \{H_{Adv}\} \) only will be accepted if it reflects an input value with EXEC = 1, as expected by \( \forall \cdot \). In VAPE’s implementation O is stored in region OR, and S in region ER. Moreover, given Theorem 3, 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 Case2 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 Case1 into three sub-cases:

Case1.1 Adv does not follow Cond1-Cond3. The only way for Adv to produce \( (H_{RA_{dv}}, O_{RA_{dv}}) \) in this case is to not call XProve, i.e., by directly guessing \( H \).

Case1.2 Adv follows Cond1-Cond3 but does not execute \( S \) between \( t_{req} \) and \( t_{verif} \). Instead, it produces \( (H_{RA_{dv}}, O_{RA_{dv}}) \) by calling:

\[
O_{RA_{dv}} \equiv XAtomicExec_{\text{PoX}}(ER_{RA_{dv}}, t_{req} \rightarrow t_{verif}) \tag{13}
\]

where \( ER_{RA_{dv}} \) is a memory region different from the one specified by \( \forall \text{rf} \) on XRequest (AdvPoX can do this by modifying METADATA to different values of \( ER_{min} \) and \( ER_{max} \) before calling XAtomicExec).

Case1.3 Similar to Case1.2, but \( ER_{RA_{dv}} \) is the same region specified by \( \forall \text{rf} \) on XRequest containing a different executable \( S_{RA_{dv}} \).

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 VAPE’s XProve function is implemented by using VRASED’s SW-Att without any modification. SW-Att covers memory regions \( MR \) (challenge memory) and \( AR \) (attested region). Hence, VAPE instantiates these memory regions as:

1. \( MR = \text{Chal} \);
2. \( ER \subset AR \);
3. \( OR \subset AR \);
4. \( METADATA \subset AR \);

Doing so ensures that all sensitive memory regions used by VAPE are included among the inputs to VRASED’s attestation. Let \( X (t) \) denote the content in memory region \( X \) at time \( t \). AdvRA can then be constructed using AdvPoX as follows:

1. AdvRA receives Chal from the challenger in step (2) of RA security game of Definition 7.
2. At arbitrary time \( t \), AdvRA has 3 options to write \( AR(t) = AR_{RA_{dv}} \) and call AdvPoX:
   
   (a) Modify \( ER(t) \neq S \) or \( OR(t) \neq O \) or \( METADATA(t) \neq METADATA_{\forall \text{rf}} \). It then calls AdvPoX in Case1.1.
   
   (b) Modify \( ER \) to be different from the range chosen by \( \forall \text{rf} \). Therefore, \( METADATA(t) \neq METADATA_{\forall \text{rf}} \). It then calls AdvPoX in Case1.2.
   
   (c) Modify \( ER(t) \) to be different from \( S \). It then calls AdvPoX in Case1.3.

In any of these options, AdvRA will produce \( (H_{RA_{dv}}, O_{RA_{dv}}) \), such that \( \text{XVerify}(H_{RA_{dv}}, O_{RA_{dv}}, S, \text{Chal}, \cdot) = 1 \) with non-negligible probability.

3. AdvRA replies to the challenger with the pair \( (M, H_{RA_{dv}}) \), where \( M \) corresponds to the values of \( S, O \) and METADATA \( \forall \text{rf} \), matching \( H_{RA_{dv}} \) and \( O_{RA_{dv}} \) generated by AdvPoX. By construction \( M \neq AR_{RA_{dv}} = AR(t) \), as required by Definition 7.

4. Challenger will accept \( (M, H_{RA_{dv}}) \) with the same non-negligible probability that AdvPoX has of producing \( (H_{RA_{dv}}, O_{RA_{dv}}) \) such that \( \text{XVerify}(H_{RA_{dv}}, O_{RA_{dv}}, S, \text{Chal}, \cdot) = 1 \).

\[\square\]

C Executable Limitations

We now discuss the limitations of our approach on the executable types.

Shared libraries. In order to produce a valid proof, \( \forall \text{rf} \) must ensure that execution of \( S \) does not depend on external code located outside its execution range \( ER \) (e.g., shared libraries). A call to such code would violate LTL 4 resulting in EXEC = 0 during the HMAC computation. One possible way to support this type of executable is to transform it into a self-contained executable by statically linking all dependencies during the compilation time. Another is to appropriately set \( ER \) to cover all external code used by \( S \).

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

Interrupts. Our notion of successful execution in Section 5.1 prohibits an interrupt to happen during \( S \)’s execution. This limitation can be problematic especially for interrupt-driven programs such as the ones in real-time systems. Nonetheless, simply allowing interrupts to happen during the execution may result in attacks that allow malware to modify intermediate execution results in data memory and consequently influence the execution output. One possible way to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) immutable from the start of execution till the end of attestation and (2) included in the attested memory range during the attestation process. \( \forall \text{rf} \) then can determine whether an interrupt that may have happened during the execution is malicious by inspecting all interrupt handlers from the proof of execution.

D Software Transformation

Recall that our notion of successful execution (in Section 5.1) requires the function’s entry point to be at the first instruction and the exit point to be at the last instruction. In this section, we discuss an efficient way to transform arbitrary software (besides the ones in Appendix C) implementing a function to conform with this requirement.

Line 10-17 of Figure 8 shows an (partial) implementation of the ReadSensor function described in Section 7.3. This implementation, when converted to an executable, does
1 #define P4IN (*volatile unsigned char *) 0x001C
2 #define P4OUT (*volatile unsigned char *) 0x001D
3 #define P4DIR (*volatile unsigned char *) 0x001E
4 #define P4SEL (*volatile unsigned char *) 0x001F
5 #define BIT4 (0x0010)
6 #define MAXTIMINGS 85
7
8 #define OR 0xEEE0 // OR is in AR
9 #define HIGH 0x1
10 #define LOW 0x0
11 #define INPUT 0x0
12 #define OUTPUT 0x1
13
14 __attribute__((section(".exec.entry"), naked)) void ReadSensorEntry() {
  ReadSensor();
  __asm__ volatile("br #__exec_leave""
	"");
}

16 __attribute__((section(".exec.body"))) void digitalRead() {
  if(PIN & BIT4) return HIGH;
  else return LOW;
}
17
18 __attribute__((section(".exec.body"))) void digitalWrite(uint8_t val) {
  if(val == LOW)
    POUT &= ~BIT4;
  else
    POUT |= BIT4;
}
19
20 __attribute__((section(".exec.body"))) void pinMode(uint8_t val) {
  if(val == INPUT)
    P3DIR &= ~BIT4;
  else if(val == OUTPUT)
    P3DIR |= BIT4;
}
21
22 // Tell the sensor that we are about to read
23 digitalWrite(HIGH);
24 delayMicroseconds(250);
25 pinMode(OUTPUT);
26 digitalWrite(LOW);
27 delayMicroseconds(250);
28 pinMode(HIGH);
29 pinMode(5); // Read the sensor’s value
30 for(i = 0; i < MAXTIMINGS; i++) {
  counter = 0;
  while(digitalRead() == laststate) {
    counter++;
  }
}
31 // Copy the reading to OR
32 memcpy(OR, data, 5);
33
34 __attribute__((section(".exec.exit"), naked)) void ReadSensorExit() {
  __asm__ volatile("ret""
	"");
}
35
(a) Fire Sensor’s code written in C

1
2 SECTIONS
3 { 
4  .text :
5  { 
6    .(exec.entry) = ALIGN(2);
7    .(exec.body) = ALIGN(2);
8    PROVIDE(.exec_leave =
9    {.exec.exit})
10  } > REGION_TEXT
11 }

(b) Linker script

Figure 8: Code snippets for (a) fire sensor described in Section 7.3 (b) linker script
not guarantee VAPE’s executable requirement since the compiler may choose to place one of its sub-functions, instead of ReadSensor, to the entry and/or exit points of the executable. One obvious way to fix this issue is to implement all of its sub-functions as inline functions; however, such approach may be inefficient as in this example it will create multiple duplicate code for the same sub-functions (e.g., digitalWrite) inside the executable.

Instead, we created the dedicated functions for the entry (Line 1-4) and exit (Line 6-8) points, and assign those functions to separated executable sections – “.exec.entry” for the entry and “.exec.exit” for the exit. Then, we labeled all sub-functions used by ReadSensor as well as ReadSensor itself to the same section – “.exec.body” – and modified the MSP430 linker to place “.exec.body” between “.exec.entry” and “.exec.exit” sections. The modified linker script is shown in Figure 8b. This way, we ensure that the entry and exit function locate at the beginning and the end of the executable, respectively, and thus the resulting executable conforms with VAPE’s requirement.