CASU: Compromise Avoidance via Secure Update for Low-end Embedded Systems

Ivan De Oliveira Nunes
Rochester Institute of Technology

Sashidhar Jakkamsetti
University of California Irvine

Youngil Kim
University of California Irvine

Gene Tsudik
University of California Irvine

ABSTRACT

Guaranteeing runtime integrity of embedded system software is an open problem. Trade-offs between security and other priorities (e.g., cost or performance) are inherent, and resolving them is both challenging and important. The proliferation of runtime attacks that introduce malicious code (e.g., by injection) into embedded devices has prompted a range of mitigation techniques. One popular approach is Remote Attestation ($\mathcal{R}$A), whereby a trusted entity (verifier) checks the current software state of an untrusted remote device (prover). RA yields a timely authenticated snapshot of prover state that verifier uses to decide whether an attack occurred.

Current RA schemes require verifier to explicitly initiate $\mathcal{R}$A, based on some unclear criteria. Thus, in case of prover's compromise, verifier only learns about it late, upon the next $\mathcal{R}$A instance. While sufficient for compromise detection, some applications would benefit from a more proactive, prevention-based approach. To this end, we construct CASU: Compromise Avoidance via Secure Updates. CASU is an inexpensive hardware/software co-design enforcing: (i) runtime software immutability, thus precluding any illegal software modification, and (ii) authenticated updates as the sole means of modifying software. In CASU, a successful $\mathcal{R}$A instance serves as a proof of successful update, and continuous subsequent software integrity is implicit, due to the runtime immutability guarantee. This obviates the need for $\mathcal{R}$A in between software updates and leads to unobtrusive integrity assurance with guarantees akin to those of prior $\mathcal{R}$A techniques, with better overall performance.

1 INTRODUCTION

Over the past two decades, Internet-of-Things (IoT) devices and Cyber-Physical Systems (CPS) have become very popular. They are deployed in many everyday settings, including both private (e.g., homes, offices, and factories) and public (e.g., cultural, entertainment, and transportation) spaces. They are also widely used in farming, industrial, and vehicular automation. These devices often collect sensitive information and perform safety-critical tasks. Also, in many cases, they are both interconnected and connected to the global Internet. They are usually implemented atop low-end microcontroller units (MCUs) that have very stringent cost, size, and energy constraints, and unlike their higher-end counterparts, have no (or few) security features. It is thus not at all surprising that these embedded devices (sensors, actuators, and hybrids) have become attractive attack targets.

In particular, code injection attacks [1–4] represent a real and prominent threat to low-end devices. Embedded systems software is mostly written in C, C++, or Assembly – languages that are very prone to errors. Code injection attacks exploit these errors to cause buffer overflows and inject malicious code into the existing software or somewhere else in the device memory.

Some previous results considered such attacks in low-end devices and proposed security techniques such as Remote Attestation ($\mathcal{R}$A) [5–10], as well as proofs of remote software updates and memory erasure [11–13]. $\mathcal{R}$A aims to detect compromise by authenticated measurement of the device's current software state. However, it has considerable runtime costs since it requires computing a cryptographic function (usually, a Message Authentication Code (MAC)) over the entire software. A recent result, *RATA* [14], minimized the cost of $\mathcal{R}$A by measuring a constant-size memory region that reflects the time of last software modification (legal or otherwise). *RATA* achieved that by introducing a hardware security monitor that securely logs each modification time to that region.

Regardless of their specifics, $\mathcal{R}$A techniques only detect code modifications **after the fact**. They cannot prevent them from taking place. Hence, there could be a sizeable window of time between the initial compromise and the next $\mathcal{R}$A instance when the compromise would be detected.

To this end, the goal of this paper is to take a more proactive, prevention-based approach to avoid potential compromise. It constructs CASU: Compromise Avoidance via Secure Update, which consists of two main components. First is a simple hardware security monitor that is formally verified. It performs two functions: (1) blocks all modifications to the specific program memory region where the software resides, and (2) prevents anything stored outside that region from executing. It runs independently from (in parallel with) the MCU core, without modifying the latter. This thwarts all code injection attacks. However, it is unrealistic to prohibit all modifications to program memory, since genuine software updates need to be installed during the device's lifetime. Otherwise, the software could be housed in ROM or the entire device would function as an ASIC (Application Specific Integrated Circuit). Therefore, CASU second component is a secure remote software update scheme.

The key benefit of CASU is maintaining constant software integrity without repeated $\mathcal{R}$A measurements while allowing genuine secure software updates. Specifically, it guarantees that, between any two successive secure updates, device software is immutable. However, the device liveness can be ascertained at any time by repeating the latest update, which essentially represents $\mathcal{R}$A.

The intended contributions of CASU are:

(1) A tiny formally verified hardware monitor that guarantees benign (authorized) software immutability and prevents the execution of any unauthorized code.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10.
https://doi.org/10.1145/3508352.3549450

(2) A scheme to enable secure software updates when authorized by a trusted 3rd party.
(3) An open-source CASU prototype built atop a commodity low-end MCU to demonstrate its low cost and practicality.

2 PRELIMINARIES

2.1 Targeted Devices

This paper focuses on CPS/IoT sensors and actuators (or hybrids thereof) with low computing power. These are some of the smallest and weakest devices based on ultra-low-power single-core MCUs with only a few KBytes of memory. Two prominent examples are Atmel AVR ATmega [15] and TI MSP430[16], with 8- and 16-bit CPUs respectively, typically running at 1-16MHz clock frequencies, with ≈ 64 KBytes of addressable memory. Figure 1 shows a typical architecture of such an MCU. It includes a CPU core, a Direct Memory Access (DMA) controller, and an interrupt control logic connected to the memory via a bus. DMA is a hardware controller that can read/write to memory in parallel with the core. Main memory contains several regions: Interrupt Vector Table (IVT), program memory (PMEM), read-only memory (ROM), data memory (DMEM or RAM), and peripheral memory. IVT stores pointers to the Interrupt Service Routines (ISRs), where the execution jumps when an interrupt occurs; it also contains the Reset Vector pointer from where the core starts to execute, after a reboot. Application software is installed in PMEM and it uses DMEM for its stack and heap. ROM contains the bootloader and/or any immutable software hard-coded at manufacturing time.

MCUs usually run software atop "bare metal" and execute instructions in place, i.e., directly from PMEM. They have neither memory management units (MMUs) to support virtualization, nor memory protection units (MPUs) for isolating memory regions. Therefore, privilege levels and isolation regimes used in higher-end devices and generic trusted execution environments (e.g., ARM TrustZone [17] or Intel SGX [18]) are not viable.

NOTE: Our initial implementation of CASU uses MSP430 MCU, a common platform for low-end embedded devices. One important factor in this choice is the public availability of an open-source MSP430 MCU design – OpenMSP430 [19]. Nonetheless, CASU is readily applicable to other low-end MCUs of the same class.

2.2 Remote Attestation & VRASED

$\mathcal{R}$A, mentioned above, allows a trusted entity (verifier = $\mathcal{V}$rf) to remotely measure current memory contents (e.g., software) of an untrusted embedded device (prover = $\mathcal{P}$rv). $\mathcal{R}$A is usually realized as a simple challenge-response protocol:

(1) $\mathcal{V}$rf sends an $\mathcal{R}$A request with a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv.
(2) $\mathcal{P}$rv receives the request and computes an authenticate integrity check over its software memory region and $\mathcal{C}$hal. The memory region can be either pre-defined or explicitly specified in the $\mathcal{R}$A request.
(3) $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
(4) $\mathcal{V}$rf verifies the result and decides if $\mathcal{P}$rv is in a valid state.

Although several $\mathcal{R}$A techniques for low-end devices have been proposed, only very few offer any concrete (provable) security guarantees. The latter include SIMPLE[8], *VRASED* [7], and a variant of SANCUS[6]. While SIMPLE, as its name suggests, is simple, it is a purely software-based $\mathcal{R}$A technique (meaning that no hardware modifications are needed) that only protects against remote attacks and does not support DMA. Whereas, SANCUS is a purely hardware-based $\mathcal{R}$A technique which, though very fast, incurs a significant hardware cost over the baseline MCU.

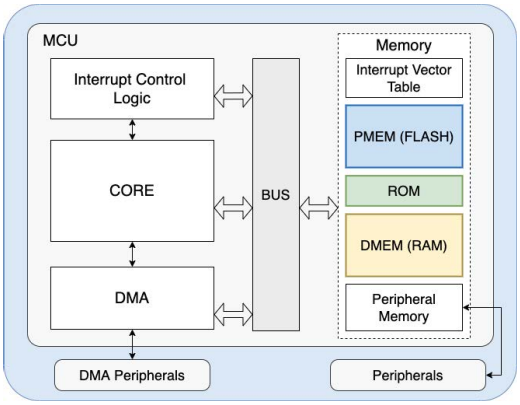


Figure 1: System architecture of a typical low-end MCU.

VRASED [7] is a formally verified hybrid (hardware/software) $\mathcal{R}$A design comprising verified hardware and software sub-modules. The software sub-module, which is immutable (stored in ROM), implements the authenticated integrity function computed over some "Attested Region" (AR) of $\mathcal{P}$rv memory (usually in PMEM). Meanwhile, its hardware component assures that its software counterpart executes securely and that no function of the $\mathcal{R}$A secret key ($\mathcal{K}$) is ever leaked. The authenticated integrity function is realized with a formally verified HMAC implementation from the HACL* cryptographic library [20] used to compute:

$$H = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), AR) \quad (1)$$

where $KDF(\mathcal{K}, \mathcal{C}\text{hal})$ is a one-time key derived from the received $\mathcal{C}$hal and $\mathcal{K}$ using a key derivation function.

NOTE: CASU uses *VRASED* to verify the update request before it installs the new software on the device. Specifically, CASU invokes *VRASED* to compute equation 1 on the new software and checks whether H matches an authentication token sent in the update request. Consequently, CASU update verification inherits the security properties of *VRASED*.

2.3 TOCTOU Attacks & TOCTOU-Security

All $\mathcal{R}$A techniques share a common limitation: they yield no information about the state of $\mathcal{P}$rv software during the time **between** two consecutive $\mathcal{R}$A instances. Consequently, it is impossible to detect the past presence of transient malware that: (1) infected $\mathcal{P}$rv, (2) remained active for a while, and (3) at some later time erased itself and restored $\mathcal{P}$rv software to its "good" state. This holds as long as (1)-(3) occur between two successive $\mathcal{R}$A instances. This attack type is referred to as *Time-Of-Check Time-Of-Use* (TOCTOU).

One recent technique, *RATA* [14], mitigates TOCTOU attacks with a minimal additional hardware component that securely logs the time of the last PMEM modification to a protected memory region called Latest Modification Time (LMT) that can not be modified by

any software. LMT is then covered by the $\mathcal{R}$A function. Therefore, an $\mathcal{R}$A response captures both the current software state of $\mathcal{P}$rv and the time of change to that state. Furthermore, *RATA* minimizes the computational cost of $\mathcal{R}$A for $\mathcal{P}$rv, since, instead of attesting its entire software, it suffices for $\mathcal{P}$rv to attest just the LMT. This way, instead of computing a MAC over the entire PMEM, $\mathcal{P}$rv computes it over a fixed-size (32-byte) *LMT* region.

NOTE: In this paper, unlike *RATA*, CASU actively **prevents** any modification to PMEM at runtime, unless it is a securely and causally authorized (by the trusted $\mathcal{V}$rf) software update.

3 CASU SCHEME & ASSUMPTIONS

3.1 Basics

Similar to the typical $\mathcal{R}$A setting, CASU involves a low-end MCU ($\mathcal{P}$rv) and verifier ($\mathcal{V}$rf). The latter is a trusted higher-end device, e.g, a laptop, a smartphone, a smart home gateway, or a device manufacturer's back-end server. $\mathcal{V}$rf is responsible for initiating each software update request, verifying whether the update was successful, and keeping track of the latest successfully confirmed software update. We assume a single $\mathcal{V}$rf for a given $\mathcal{P}$rv. Also, $\mathcal{P}$rv and $\mathcal{V}$rf are assumed to share a master secret key ($\mathcal{K}$) installed on $\mathcal{P}$rv at manufacturing time. Our discussion focuses on the symmetric key setting, which is more practical for low-end MCUs. Nonetheless, the use of public-key cryptography is possible with some cosmetic changes to CASU, provided that $\mathcal{P}$rv has sufficient computing capabilities[1].

3.2 Secure Update Overview

At the time of its initial deployment, $\mathcal{V}$rf is assumed to know the software state ($\mathcal{S}_{\text{old}}$) of $\mathcal{P}$rv. When $\mathcal{V}$rf later wishes to update this software, it issues an update request, denoted by $\textbf{Update}^{\mathcal{V}\text{rf}}$, to $\mathcal{P}$rv. This request carries the new software $\mathcal{S}_{\text{new}}$ and a fresh authentication token ATok, based on $\mathcal{S}_{\text{new}}$.

When $\mathcal{P}$rv receives an $\textbf{Update}^{\mathcal{V}\text{rf}}$, $\mathcal{S}_{\text{old}}$ invokes CASU, which handles the update process in two steps: (1) $\textbf{Auth}^{\mathcal{P}\text{rv}}$ verifies that ATok is a fresh and timely token that corresponds to $\mathcal{S}_{\text{new}}$, and (2) if the first step succeeds, $\textbf{Install}^{\mathcal{P}\text{rv}}$ replaces $\mathcal{S}_{\text{old}}$ with $\mathcal{S}_{\text{new}}$ and generates an authenticated acknowledgment (AAck). At this point, CASU terminates and control is given to $\mathcal{S}_{\text{new}}$ which must send AAck to $\mathcal{V}$rf.

Upon receiving AAck, $\mathcal{V}$rf executes the $\textbf{Verify}^{\mathcal{V}\text{rf}}$ procedure to check whether the AAck is a valid confirmation for the outstanding $\textbf{Update}^{\mathcal{V}\text{rf}}$. If no AAck is received, or if AAck verification fails, $\mathcal{V}$rf assumes a failed update. Figure 2 illustrates the interaction between $\mathcal{V}$rf and $\mathcal{P}$rv. Protocol details are described in Section 4 below.

3.3 Adversary Model

We consider an adversary, $\mathcal{A}$dv, that controls the entire memory state of $\mathcal{P}$rv, including PMEM (flash) and DMEM (RAM). It can attempt to write, read or execute any memory location. It can also attempt to remotely launch code injection attacks to modify $\mathcal{P}$rv software. It may also divert the execution control-flow to ignore update requests, as well as attempt to extract any $\mathcal{P}$rv secrets or forge update confirmations.

Furthermore, $\mathcal{A}$dv can configure DMA controllers on $\mathcal{P}$rv to read/write to any part of the memory while bypassing the CPU. It can induce interrupts in an attempt to pause the update procedure, modify any part of the old or new software versions, or cause inconsistencies or race conditions. It might also eavesdrop on, and interfere, with network traffic between $\mathcal{V}$rf and $\mathcal{P}$rv, in a typical Dolev-Yao manner [21].

As common in most related work, physical attacks requiring adversarial presence are considered out of scope. This includes both non-invasive and invasive physical attacks. The former describes attacks whereby $\mathcal{A}$dv physically reprograms $\mathcal{P}$rv software using direct/wired interfaces, such as USB/UART, SPI, or I2C. The latter refers to inducing hardware faults, modifying code in ROM, extracting secrets via physical side-channels, and tampering with hardware. Protection against non-invasive attacks can be obtained via well-known features, such as a secure boot. Whereas, protection against invasive attacks can be obtained via standard tamper-resistant techniques [22].

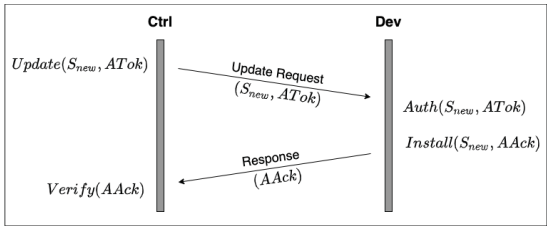


Figure 2: CASU **Secure Update Protocol.**

4 CASU DESIGN

One of CASU main features is the prevention of all unauthorized software modifications to $\mathcal{P}$rv software. As mentioned earlier, the former can be trivially achieved by making all $\mathcal{P}$rv software read-only, or by making $\mathcal{P}$rv an ASIC. However, this precludes all benign (authorized) updates. Therefore, it is essential to have a secure update mechanism. The term "authorized" refers to software installed on $\mathcal{P}$rv physically at manufacture or deployment time, as well as each subsequent version installed via update request by $\mathcal{V}$rf.

From $\mathcal{V}$rf perspective, CASU guarantees that, once installed, authorized software on $\mathcal{P}$rv remains unchanged until the next $\mathcal{V}$rf-initiated successful secure update. This is achieved via three features:

(1) *Authorized Software Immutability:* Except via a secure update (implemented within CASU trusted code), authorized software cannot be modified.
(2) *Unauthorized Software Execution Prevention:* Only the memory containing the (immutable) authorized software is executable.
(3) *Secure Update:* $\mathcal{V}$rf is the only entity that can authenticate to $\mathcal{P}$rv to install new software. After an update, the previous version of the installed software is no longer authorized.

The first two features are realized by a hardware module, CASU-*HW*, that runs in parallel with the CPU. It monitors a few CPU hardware

[1] In case of MSP430, based on our experimental attempts, neither generating nor even verifying public key signatures is viable.

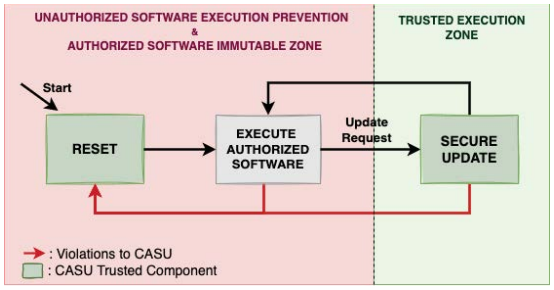


Figure 3: CASU **Software Execution Flow.**

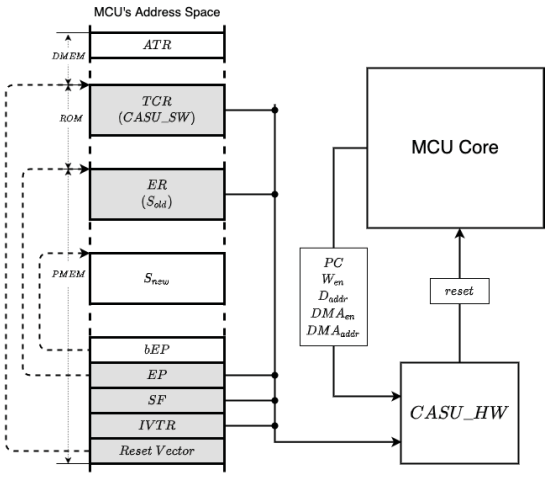


Figure 4: CASU **System Architecture.**

signals and triggers an MCU reset if any violation is detected. The third feature is realized by a trusted code base (TCB), CASU-*SW*, that extends *VRASED* to authenticate incoming update requests containing new software to be installed ($\mathcal{S}_{\text{new}}$) and an authorization token (ATok) that must be issued by $\mathcal{V}$rf using the key $\mathcal{K}$ pre-shared with CASU module within $\mathcal{P}$rv. If ATok matches $\mathcal{S}_{\text{new}}$, then CASU-*SW* installs $\mathcal{S}_{\text{new}}$ on $\mathcal{P}$rv and produces an authenticated AAck, attesting to $\mathcal{V}$rf that a successful update occurred on $\mathcal{P}$rv.

Figure 3 depicts CASU software execution flow. After each boot or reset, it executes authorized software that was previously installed (either physically or via CASU Secure Update). In this state, CASU-*HW* ensures software immutability and execution prevention of anything else. However, when an update request is received, CASU-*SW* must be invoked to securely apply the update and reconfigure CASU-*HW* to protect the memory region where $\mathcal{S}_{\text{new}}$ is installed. Note that the update cannot be performed without invoking CASU-*SW* due to the immutability guarantee.

Table 1 summarizes MCU hardware signals and memory regions relevant to CASU. Figure 4 illustrates the CASU architecture: (1) CASU-*HW* prevents modification of memory regions in gray and prevents execution of all other memory, while (2) CASU-*SW* resides in the ROM; it contains a bootloader and subroutines related to secure update. We describe these features in detail in the rest of this section.

Table 1: Notation Summary

Notation	Description
PC	Program Counter, points to the current instruction being executed
W_{en}	1-bit signal that indicates if MCU core is writing to memory
D_{addr}	Memory address where the MCU core is currently accessing
DMA_{en}	1-bit signal that indicates if DMA is active
DMA_{addr}	Memory address being accessed by DMA, when active
$reset$	Signal that reboots the MCU when set to logic '1'
TCR	Trusted Code Region, a fixed ROM region storing CASU-*SW*
ER	Executable Region, a configurable memory region where authorized software is stored; $ER = [ER_{min}, ER_{max}]$, where ER_{min} and ER_{max} are the boundaries of ER
EP	Executable Pointer, a fixed memory region storing current values of ER_{min} and ER_{max}
bEP	Buffer Executable Pointer, a fixed memory location used to save the boundaries of the memory region storing new software $\mathcal{S}_{\text{new}}$.
ATR	Fixed memory buffer from which $\textbf{Auth}^{\mathcal{P}\text{rv}}$ reads ATok and also where $\textbf{Install}^{\mathcal{P}\text{rv}}$ outputs AAck
$IVTR$	Reserved memory region for the MCU's IVT
SF	Fixed memory region where $Status$ flag is stored; $Status$ is used by CASU-*SW* for consistency.

4.1 CASU-*HW*: Hardware Security Monitor

CASU-*HW* monitors PC, W_{en}, D_{addr}, DMA_{en}, DMA_{addr} to detect illegal writes or execution. When a *violation* is detected, CASU-*HW* activates the $reset$ signal. To simplify notation when describing CASU-*HW* properties, we define the following macro:

$$Mod\_Mem(i) \equiv (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$$

i represents a memory address. $Mod\_Mem\ (i)$ is true whenever the MCU core or the DMA is writing to i. When representing a write within some contiguous memory region (with multiple addresses) $M = [M_{min}, M_{max}]$, we "abuse" the notation as $Mod\_Mem\ (M)$. To denote that a write has occurred within one of the multiple contiguous memory regions, e.g., when a write happens to some address within M_1 or M_2, we say $Mod\_Mem\ (M_1, M_2)$.

4.1.1 **Authorized Software Immutability.** Software authorized by CASU, including any ISRs, is located in the contiguous memory segment *ER*. The pointer *EP* stores the boundaries that define *ER*, i.e., ER_{min} and ER_{max}. CASU-*HW* monitors *EP* to locate the currently authorized software and enforce its rules based on this region. Write attempts to *EP* are also monitored and only allowed when performed by CASU-*SW*, preventing malicious changes to *EP* that could misconfigure the definition *ER*, leading CASU-*HW* to enforce protections based on the incorrect region. *ER* is configurable to give CASU-*SW* flexibility to change the location and size of authorized software, instead of fixing $\mathcal{S}_{\text{new}}$ to the same location and size of $\mathcal{S}_{\text{old}}$, as software versions vary in size. CASU-*HW* also protects memory regions *SF* and *IVTR*. *SF* is used during a secure update, described in Section 4.2. Since ISRs are a part of *ER*, IVT must be protected to maintain the integrity of interrupt handling during authorized software execution.

Incidentally, Authorized Software Immutability also prohibits self-modifying code, i.e., code in *ER* writing to *ER*, to prevent code injection attacks within *ER*.

4.1.2 **Unauthorized Software Execution Prevention.** Only authorized software (located in *ER*) or CASU-*SW* (located in *TCR*) are allowed to execute on $\mathcal{P}$rv. Since *ER* is configurable via *EP*, after a secure update, CASU-*SW* re-configures *EP* to allow execution from the new *ER* location.

Authorized Software Immutability:

$$[Mod\_Mem(ER, EP, SF, IVTR) \wedge (PC \notin TCR)] \rightarrow reset \quad (2)$$

Unauthorized Software Execution Prevention:

$$[(PC \notin ER) \wedge (PC \notin TCR)] \rightarrow reset \quad (3)$$

Figure 5: CASU-*HW* **Security Properties.**

4.1.3 CASU-*HW* **Properties Formally.** Figure 5 formalizes the aforementioned CASU-*HW* security properties using proportional logic. Note that these properties must hold at all times. Equation 2 states that any modification to *ER*, *EP*, *SF*, and *IVTR*– when a program other than CASU-*SW* ($PC \notin TCR$) is executing – causes a *reset*. Equation 3 states that MCU cannot execute programs other than those in *ER* and *TCR*. If *PC* points to any other memory location, the MCU is reset.

4.2 CASU Secure Update

Recall (from Section 3.2) that CASU Secure Update implements: (**Update**$^{\mathcal{V}\text{rf}}$, **Verify**$^{\mathcal{V}\text{rf}}$) on $\mathcal{V}$rf and (**Auth**$^{\mathcal{P}\text{rv}}$, **Install**$^{\mathcal{P}\text{rv}}$) on $\mathcal{P}$rv. At a high level, there are two ways of implementing it on $\mathcal{P}$rv.

(1) Download $\mathcal{S}_{\text{new}}$ to DMEM (RAM), i.e., the stack or heap of the current software ($\mathcal{S}_{\text{old}}$), and invoke **Auth**$^{\mathcal{P}\text{rv}}$. If it succeeds, **Install**$^{\mathcal{P}\text{rv}}$ overwrites *ER* with $\mathcal{S}_{\text{new}}$ and updates *EP*. This is problematic, because, if a reset occurs in the middle of **Install**$^{\mathcal{P}\text{rv}}$ execution, then *ER* containing $\mathcal{S}_{\text{old}}$ would be partially overwritten and $\mathcal{S}_{\text{new}}$ in the DMEM would be lost as a consequence of the reset. This would leave $\mathcal{P}$rv software in a corrupted state.

(2) Download $\mathcal{S}_{\text{new}}$ to PMEM (flash) and invoke **Auth**$^{\mathcal{P}\text{rv}}$. If **Auth**$^{\mathcal{P}\text{rv}}$ succeeds, **Install**$^{\mathcal{P}\text{rv}}$ updates *EP* to the location where $\mathcal{S}_{\text{new}}$ resides. This is generally safer since $\mathcal{S}_{\text{new}}$ and $\mathcal{S}_{\text{old}}$ reside in two separate flash memory regions. If the installation is interrupted by a reset, CASU-*SW* can re-invoke **Install**$^{\mathcal{P}\text{rv}}$ to complete the installation. However, this requires $\mathcal{P}$rv PMEM to be sufficiently large to accommodate both $\mathcal{S}_{\text{new}}$ and $\mathcal{S}_{\text{old}}$, i.e., at least double the size of *ER*. We believe that this is a realistic assumption. The size of flash memory on our targetted devices is at least 8*KB*, whereas the typical binary size is usually under 2*KB*.

Construction 1 shows the whole scheme. Recall that CASU-*SW* is immutable (being in ROM). Its functionality is described below.

4.2.1 **Update**$^{\mathcal{V}\text{rf}}$. Secure update requires for any software $\mathcal{S}_{\text{new}}$ to be installed on $\mathcal{P}$rv to adhere to the following format $\mathcal{S}_{\text{new}} := (L_{\mathcal{S}_{\text{new}}} || V_{\mathcal{S}_{\text{new}}} || N_{\mathcal{S}_{\text{new}}} || BIN_{\mathcal{S}_{\text{new}}} || IVT_{\mathcal{S}_{\text{new}}})$, where $L_{\mathcal{S}_{\text{new}}}, V_{\mathcal{S}_{\text{new}}}, N_{\mathcal{S}_{\text{new}}}$ is the $\mathcal{S}_{\text{new}}$ *header* consisting of its size, version number, and a random nonce, respectively. $BIN_{\mathcal{S}_{\text{new}}}$ is the $\mathcal{S}_{\text{new}}$ binary in byte-code that mandatorily includes a **download** and **acknowledge** subroutine that accepts future update requests and replies acknowledgment message back to $\mathcal{V}$rf. $IVT_{\mathcal{S}_{\text{new}}}$ is the IVT of $\mathcal{S}_{\text{new}}$ that needs to be overwritten to *IVTR* region so that MCU knows where to jump into the new software when an interrupt is triggered. Another requirement is that $V_{\mathcal{S}_{\text{new}}}$ should always be greater than the version number of the current (or old) software on $\mathcal{P}$rv. This avoids replay attacks that attempt to trick $\mathcal{P}$rv into installing an old software version that contains vulnerabilities. In case $\mathcal{V}$rf wishes to revert to an older version (e.g., due to later-discovered bugs in $\mathcal{S}_{\text{new}}$), it must issue a brand new update request with the older-version software, though with a **new version** number.

$\mathcal{V}$rf, by invoking **Update**$^{\mathcal{V}\text{rf}}$, computes ATok using equation 4 and sends ($\mathcal{S}_{\text{new}}$, ATok) to $\mathcal{P}$rv.

4.2.2 **Auth**$^{\mathcal{P}\text{rv}}$. When $\mathcal{P}$rv receives **Update**$^{\mathcal{V}\text{rf}}$ with $\mathcal{S}_{\text{new}}$ and ATok, the current **download** subroutine on $\mathcal{S}_{\text{old}}$ in *ER* accepts and downloads $\mathcal{S}_{\text{new}}$ to an available PMEM slot. It then writes the pointers to $\mathcal{S}_{\text{new}}$ to *bEP*, buffer Executable Pointer, in PMEM, and writes ATok to *ATR*. This **download** subroutine should not be a part of CASU-*SW*, as exposing network interfaces directly to trusted parts of the device is hazardous and may result in the exploitation of unknown vulnerabilities in it, leading to key leakage. Hence, even though *ER* is untrusted, it should be the one receiving the request, because even if it fails to receive or chooses to not call **Auth**$^{\mathcal{P}\text{rv}}$, then AAck is not generated/sent, which is a clear indication to $\mathcal{V}$rf that the update was unsuccessful.

To securely verify that $\mathcal{S}_{\text{new}}$ is a valid software to be installed on $\mathcal{P}$rv, **Auth**$^{\mathcal{P}\text{rv}}$ first checks whether the $V_{\mathcal{S}_{\text{new}}}$ is greater than the one of *ER*, i.e., V_{ER}. If the $V_{\mathcal{S}_{\text{new}}}$ is valid, it invokes *VRASED* as a subroutine to compute σ according to equation 5. If σ matches with ATok received from $\mathcal{V}$rf, then it outputs $\top$ (accept symbol) and further invokes **Install**$^{\mathcal{P}\text{rv}}$ to apply the update. Otherwise, it outputs $\perp$ (reject symbol) and returns to old software at *ER* without computing any response to be sent back to $\mathcal{V}$rf.

Note that CASU-*SW* execution is guarded by CASU-*HW* (which inherits *VRASED* hardware properties), i.e., any interrupts or DMA, or any attempts to access the key or any confidential data that CASU-*SW* generates, will be considered as a *violation* and an MCU reset will be triggered immediately. Also note that if such an abrupt reset occurs, MCU will return to the old software, and eventually $\mathcal{V}$rf has to send a new update request. In this new request, $\mathcal{V}$rf can use the same version number (but with a different nonce for maintaining freshness) because the previous update was not applied, and thus, the version number of the current software is still old.

4.2.3 **Install**$^{\mathcal{P}\text{rv}}$. Once $\mathcal{S}_{\text{new}}$ is authenticated, **Install**$^{\mathcal{P}\text{rv}}$ is invoked. This is the critical step of Secure Update. It is responsible for updating the *EP* with *bEP*, *IVTR* with $IVT_{\mathcal{S}_{\text{new}}}$ and computing authenticated acknowledgment AAck that is to be replied to $\mathcal{V}$rf. As mentioned in Section 4.2.2, if a reset occurs during any of these sub-steps, they have to be repeated from the beginning. This is because, if *EP* is updated and *IVTR* is not, vulnerabilities in old ISRs pointed to by the old *IVT* can be exploited by malware. Furthermore, if *EP* and *IVTR* are updated, yet the computation of AAck failed, $\mathcal{V}$rf assumes that the update failed and repeats the update request with the same version number (since *EP* is updated to the new software), and **Auth**$^{\mathcal{P}\text{rv}}$ will fail again. Therefore, all three sub-steps must take place atomically. To this end, CASU-*SW* uses a *Status* flag *SF* in PMEM, which it sets and unsets, before and after the completion of **Install**$^{\mathcal{P}\text{rv}}$ sub-steps, respectively.

To handle cases when a reset is triggered during **Install**$^{\mathcal{P}\text{rv}}$, the Reset Vector in *IVTR* is programmed to start executing from CASU-*SW*. This technique is analogous to having a bootloader. At boot

CONSTRUCTION 1. CASU *Secure Update scheme defined by* [**Update**$^{\mathcal{V}\text{rf}}$, **Auth**$^{\mathcal{P}\text{rv}}$, **Install**$^{\mathcal{P}\text{rv}}$, **Verify**$^{\mathcal{V}\text{rf}}$] *is realized as follows:*
– $\mathcal{K}$ is a symmetric key pre-shared between $\mathcal{V}$rf and $\mathcal{P}$rv (protected by VRASED secure architecture);

(1) **Update**$^{\mathcal{V}\text{rf}}(\mathcal{S}_{\text{new}}) \rightarrow$ ATok*:*
$\mathcal{V}$rf *generates a tuple* $T := (\mathcal{S}_{\text{new}}, \text{ATok})$*, where* $\mathcal{S}_{\text{new}}$ *is the new software and* ATok *is the accompanying authentication token, as follows:*
(a) Compiles and generates $\mathcal{S}_{\text{new}} := (L_{\mathcal{S}_{\text{new}}} || V_{\mathcal{S}_{\text{new}}} || N_{\mathcal{S}_{\text{new}}} || BIN_{\mathcal{S}_{\text{new}}} || IVT_{\mathcal{S}_{\text{new}}})$*, where* $L_{\mathcal{S}_{\text{new}}}$ *is* $\mathcal{S}_{\text{new}}$ *size,* $V_{\mathcal{S}_{\text{new}}}$ *is* $\mathcal{S}_{\text{new}}$ *version number,* $N_{\mathcal{S}_{\text{new}}}$ *is a random nonce,* $BIN_{\mathcal{S}_{\text{new}}}$ *is* $\mathcal{S}_{\text{new}}$ *binary, and* $IVT_{\mathcal{S}_{\text{new}}}$ *is* $\mathcal{S}_{\text{new}}$ IVT*, to be placed in* $IVTR$ *of* $\mathcal{P}$rv*.*
(b) Computes ATok *using equation 4 with the second operand set to:* $0||\mathcal{S}_{\text{new}}$*, where '0' is the direction indicator from* $\mathcal{V}$rf *to* $\mathcal{P}$rv*.*

$$\text{ATok} := HMAC(\mathcal{K}, 0||\mathcal{S}_{\text{new}}) \tag{4}$$

$\mathcal{V}$rf *sends* T *to* $\mathcal{P}$rv *for update.*

(2) **Auth**$^{\mathcal{P}\text{rv}}(\mathcal{S}_{\text{new}}, \text{ATok}) \rightarrow \perp / \top$*:*
Upon receiving a tuple $T := (\mathcal{S}_{\text{new}}, \text{ATok})$ *from* $\mathcal{V}$rf*,* $\mathcal{S}_{\text{new}}$ *is downloaded at memory region pointed to by* bEP *and* ATok *is written to* ATR*. Then* $\mathcal{P}$rv *does the following:*
(a) If $V_{\mathcal{S}_{\text{new}}} <= V_{ER}$*, output* $\perp$ *and return to ER; otherwise, proceed to the next step.*
(b) Computes σ *using equation 5.*

$$\sigma := HMAC(\mathcal{K}, 0||bEP) \tag{5}$$

(c) If $\sigma ==$ ATok*, output* $\top$ *and invoke* **Install**$^{\mathcal{P}\text{rv}}$*; otherwise, output* $\perp$ *and return to ER, where the current software (*$\mathcal{S}_{\text{old}}$*) resides.*

(3) **Install**$^{\mathcal{P}\text{rv}}(\mathcal{S}_{\text{new}}) \rightarrow$ AAck*:*
Upon invocation by **Auth**$^{\mathcal{P}\text{rv}}$*, or at boot time, in case* $Status$ *is equal to* 1*,* $\mathcal{P}$rv *does the following:*
(a) Sets $Status$ *to* 1 *and updates* EP *with values in* bEP*.*
(b) Updates $IVTR$ *with* $IVT_{\mathcal{S}_{\text{new}}}$*.*
(c) Computes AAck *using equation 6 and stores it at* ATR*. In equation 6 the second operand is* $1||V_{\mathcal{S}_{\text{new}}}||N_{\mathcal{S}_{\text{new}}}$*, where '1' is the direction indicator from* $\mathcal{P}$rv *to* $\mathcal{V}$rf*.*

$$\text{AAck} := HMAC(\mathcal{K}, 1||V_{\mathcal{S}_{\text{new}}}||N_{\mathcal{S}_{\text{new}}}) \tag{6}$$

(d) Sets $Status$ *to* 0 *and jumps to new ER, which is pointed to by the new value in* EP*.*
$\mathcal{P}$rv *replies to* $\mathcal{V}$rf *with* AAck *indicating successful update.*

(4) **Verify**$^{\mathcal{V}\text{rf}}(\text{AAck}) \rightarrow \perp / \top$*:*
Upon receiving AAck *from* $\mathcal{P}$rv*,* $\mathcal{V}$rf *does the following:*
(a) Computes γ *using the same equation 6.*
(b) If $\gamma ==$ AAck*, outputs* $\top$*; otherwise outputs* $\perp$*.*

time, CASU-*SW* uses *Status* to determine whether a reset occurred prior to the completion of **Install**$^{\mathcal{P}\text{rv}}$. If so, CASU-*SW* re-invokes **Install**$^{\mathcal{P}\text{rv}}$ from the beginning.

Finally, **Install**$^{\mathcal{P}\text{rv}}$ computes AAck according to equation 6 and writes it to *ATR*. After generating AAck, CASU-*SW* jumps to new *ER*. Now, it is the responsibility of the **acknowledge** subroutine in $\mathcal{S}_{\text{new}}$ to reply to $\mathcal{V}$rf with AAck.

Acknowledgment Receipt: There are two unlikely cases where $\mathcal{V}$rf may not receive AAck, after being generated by **Install**$^{\mathcal{P}\text{rv}}$. Firstly, AAck sent by $\mathcal{P}$rv being lost or corrupted in transit. In this case, upon a time-out, $\mathcal{V}$rf re-sends **Update**$^{\mathcal{V}\text{rf}}$. Since **Install**$^{\mathcal{P}\text{rv}}$ stores AAck in a dedicated region of DMEM (*ATR*), **download** in *ER* checks whether the update request has the same version number as itself and directly replies AAck to $\mathcal{V}$rf, instead of invoking **Auth**$^{\mathcal{P}\text{rv}}$ again. Secondly, a reset occurring after a successful update and before AAck is sent to $\mathcal{V}$rf. In that case, AAck is lost and, upon a timeout, $\mathcal{V}$rf needs to send a new **Update**$^{\mathcal{V}\text{rf}}$ with a new version number. The drawback of this approach is that the same update is re-applied, wasting MCU clock cycles. However, the latter case is very rare, and even if it occurs, CASU-*SW* only takes less than a second to re-install $\mathcal{S}_{\text{new}}$ (see Section 6.2).

$\mathcal{V}$rf can distinguish between these cases by first re-sending the same **Update**$^{\mathcal{V}\text{rf}}$. If there is still no response, then AAck is most likely lost due to a reset and $\mathcal{V}$rf must send a new **Update**$^{\mathcal{V}\text{rf}}$ with a new version number.

There are other ways to mitigate the aforementioned AAck issues. Rather than storing AAck in DMEM, it could be placed into a reserved memory in PMEM to ensure its persistence even if a reset occurs. Now, **download** can always reply with AAck whenever it sees a duplicate request, thus eliminating the cost of re-update. However, this approach requires an additional write to flash, which may be undesirable. Alternatively, we can use a $\mathcal{V}$rf-supplied timestamp instead of a nonce in $\mathcal{S}_{\text{new}}$ and modify **Auth**$^{\mathcal{P}\text{rv}}$ to accept duplicate requests with a more recent timestamp. This approach does not require any reserved memory (not even in DMEM). However, it incurs runtime overhead every time $\mathcal{V}$rf issues a duplicate request. Each aforementioned alternative has its own benefits and drawbacks. We leave it up to $\mathcal{V}$rf to decide which is most suitable.

Note that none of the above can result in a DoS attack due to multiple requests, because all **Update**$^{\mathcal{V}\text{rf}}$-s originate from a legit $\mathcal{V}$rf and are verified by **Auth**$^{\mathcal{P}\text{rv}}$. Moreover, **download** can check the $\mathcal{S}_{\text{new}}$ header to check if the request was already seen, discard the rest of the packets, and simply reply stored AAck to $\mathcal{V}$rf.

4.2.4 **Verify**$^{\mathcal{V}\text{rf}}$*.* Finally, if all goes well, $\mathcal{V}$rf receives an AAck and checks its validity verifies using equation 6. If either AAck is invalid, or a time-out occurs, $\mathcal{V}$rf assumes that the update failed.

Figure 6 depicts the workflow of secure updates. When $\mathcal{P}$rv comes out of reset, it starts executing CASU-*SW*. CASU-*SW* first checks whether *Status* is 1, it invokes **Install**$^{\mathcal{P}\text{rv}}$ to resume installation of already verified $\mathcal{S}_{\text{new}}$ located at *bEP*. Otherwise, it jumps

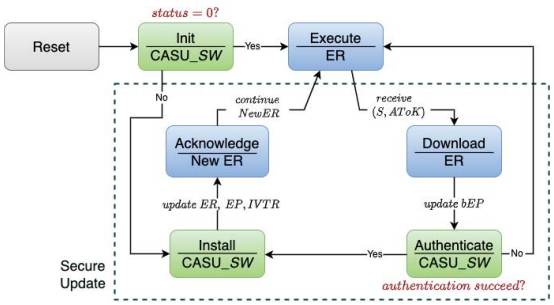


Figure 6: Secue Update Workflow: blue and green boxes indicate authorized and trusted execution routines, respectively.

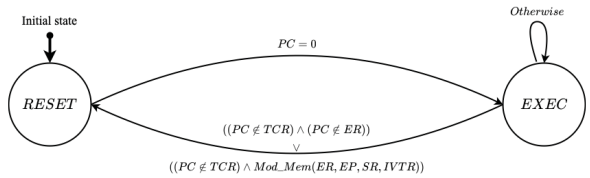


Figure 7: FSM of CASU-*HW* Verified Hardware Module.

to $\mathcal{S}_{\text{old}}$ in *ER*. Upon receiving **Update**$^{\mathcal{V}\text{rf}}$, the **download** routine in $\mathcal{S}_{\text{old}}$ accepts and downloads $\mathcal{S}_{\text{new}}$ to an available memory slot in PMEM and stores this address in *bEP*. $\mathcal{S}_{\text{old}}$ is free to complete its pending tasks before invoking **Auth**$^{\mathcal{P}\text{rv}}$ in CASU-*SW*. Once, it invokes CASU-*SW*, atomic execution of **Auth**$^{\mathcal{P}\text{rv}}$ and **Install**$^{\mathcal{P}\text{rv}}$ (if the former succeeds) begins. During **Install**$^{\mathcal{P}\text{rv}}$, if a *violation* is detected, $\mathcal{P}$rv resets and invokes CASU-*SW* with *Status* set to 1, thus invoking **Install**$^{\mathcal{P}\text{rv}}$ again. After successful completion of **Install**$^{\mathcal{P}\text{rv}}$, CASU-*SW* jumps to $\mathcal{S}_{\text{new}}$ in *ER*. Eventually, the **acknowledge** in $\mathcal{S}_{\text{new}}$ replies AAck to $\mathcal{V}$rf, and continues with its normal execution.

5 IMPLEMENTATION

5.1 CASU-*HW* Verified Hardware Module

Figure 7 presents a hardware FSM formally verified to enforce both properties of Figure 5. It is a Mealy FSM, where output is determined by both the current state and current input. This FSM takes as input the signals shown in Figure 4 and produces a single one-bit output *reset*. If *reset* is 1, the MCU core immediately resets.

There are two states in the FSM: *RESET* and *EXEC*. In *RESET*, *reset* is 1 and remains so until the FSM leaves that state; in other cases reset is 0. After a reset, as soon as *PC* reaches 0 (execution is ready to start), the FSM transitions to *EXEC*. While in *EXEC*, the FSM constantly checks for: *(1)* modifications to *ER*, *EP*, *SF*, or *IVTR*, and *(2)* execution attempts outside *ER* and *TCR*. In either case, the FSM transitions to *RESET*.

We implement the FSM using Verilog HDL and automatically translate it into Symbolic Model Verifier (SMV) language using Verilog2SMV [23] tool. Finally, we use the NuSMV Model Checker [24] to generate machine proofs showing that the FSM adheres to the properties in Figure 5.

5.2 CASU-*SW* Secure Update Routine

CASU-*SW* implements subroutines `casu_entry`, `casu_authenticate`, `casu_install`, and `casu_exit`.

`casu_entry` is the only legal entry point to CASU-*SW*; it is invoked at boot and during an update. Boot invocation is obtained by setting the IVT reset vector to `casu_entry`. `casu_entry` takes a boolean argument to test whether it was invoked at boot or by *ER* for an update. In the former case, it checks *Status* to determine whether to invoke `casu_install` in order to resume the unfinished update from the last reset. Otherwise, it calls `casu_exit`, which clears the MCU registers and jumps to the binary in *ER*. In the latter case, it invokes `casu_authenticate`. `casu_authenticate` checks for the validity of the version number of $\mathcal{S}_{\text{new}}$ at *bEP* and invokes *VRASED* software to compute HMAC. If the measurement matches ATok, `casu_install` is invoked; otherwise, it jumps to `casu_exit`. Finally, `casu_install` updates *EP*, copies the new IVT to *IVTR*, and computes and stores AAck at *ATR*. It also sets/unsets *Status* to indicate the status of installation to `casu_entry` subroutine, in case of a reset.

CASU-*SW* is implemented in C with a tiny TCB of $\approx$ 140 lines of code. It uses *VRASED* software, which is implemented using a formally verified cryptographic library, HACL* [20].

6 EVALUATION

All CASU source code and hardware verification/proofs are publicly available at [25]. CASU prototype is built on OpenMSP430 [19], an open-source implementation of TI-MSP430 [16]. We use Xilinx Vivado to synthesize an RTL description of CASU-*HW* and deploy it on the Diligent Basys3 board featuring an Artix7 FPGA.

6.1 Hardware Overhead

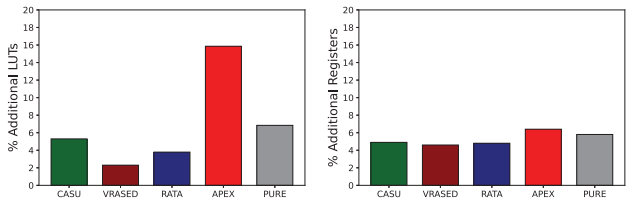
Table 2 presents CASU hardware overhead compared to unmodified OpenMSP430 and *VRASED*. Similar to prior work [5–7, 26], we consider additional Look-Up Tables (LUTs) and registers. Compared to *VRASED*, CASU only requires 3% (99) additional LUTs and 0.3% (34) additional registers.

Verification Cost: CASU was verified using a Ubuntu 18.04 LTS machine running 3.2GHz with 16GB of RAM. Table 2 shows verification time and memory. CASU requires 95 additional lines of Verilog code to enforce properties in Figure 5. The verification cost includes the verification of *VRASED* properties. The time to verify the composite design is under a second and requires 148MB of RAM.

Table 2: Hardware Overhead & Verification cost.

Architecture	Hardware		Verification			
	LUTs	Regs	LoC	#(LTLs)	Time (s)	RAM (MB)
OpenMSP430	1859	692	-	-	-	-
VRASED	1902	724	481	10	0.4	13.6
CASU (+*VRASED*)	1958	726	576	12	0.9	148

Comparison with Related Architectures: In Figure 8, we compare CASU with other low-end MCU security architectures, including *VRASED* [7], *RATA* [14], *APEX* [26], and *PURE* [11], which provide $\mathcal{R}$A-related services. However, recall that, unlike CASU, all these other architectures are reactive. As a superset of *VRASED*,



(a) Additional HW overhead (%) in Number of Look-Up Tables (b) Additional HW overhead (%) in Number of Registers

Figure 8: Hardware Overhead Comparison.

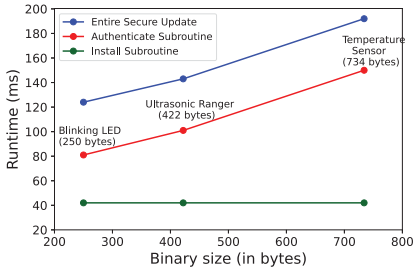


Figure 9: Runtime of CASU-*SW* **Secure Update**

CASU naturally has a higher overhead. CASU and *RATA* have similar overheads, since both monitor memory modifications. Whereas *APEX* and *PURE* enforce additional hardware properties for generating proofs of execution (*APEX*), and proofs of update, reset, erasure (*PURE*); and thus, they have a higher overhead than CASU.

6.2 Runtime for Secure Updates

The runtime of CASU-*SW* was evaluated on three sample applications: (1) Blinking LED (250 bytes of binary size) - toggles an LED every half a second, (2) Ultrasonic Ranger (422 bytes) - available at [27] - computes the distance of an obstacle from a moving object, and (3) Temperature Sensor (734 bytes) - available at [28] - measures the temperature of a room. In each case, we measured execution time of `casu_authenticate` and `casu_install` – the most time-consuming tasks dominated by HMAC computations. Results are shown in Figure 9. `casu_install` runtime is constant because it updates fixed-size memory ranges (including *EP*, *IVTR*, and *SF*) and computes HMAC on a fixed-size input. Whereas, `casu_authenticate` scales linearly with $\mathcal{S}_{\text{new}}$ size, over which HMAC is computed. The combined runtime for the worst case (temperature sensor case with 734-byte binary) is $\approx 200ms$, which we consider to be reasonable, considering that updates are infrequent.

Reserved Memory: CASU requires 32 bytes of reserved RAM for *ATR*, 8 bytes of reserved PMEM for *EP* and *bEP*, and 1 byte of PMEM for *SF*. In total, it consumes 41 bytes of additional storage.

7 RELATED WORK

Prior related work generally falls into two categories: *passive* and *active* Roots-of-Trust (RoTs).

Passive RoTs aim to detect software compromise by producing an unforgeable proof of $\mathcal{P}$rv state to $\mathcal{V}$rf. In terms of functionality, they implement the following services: (1) memory integrity verification, i.e., $\mathcal{R}$A [5–10, 29–36]; (2) verification of runtime properties, including control-flow and data-flow attestation [26, 37–44]; and (3) proofs of remote software update, erasure, and reset [11–13]. As mentioned in Section 1, they are passive in nature and do not prevent modifications. Whereas, CASU is active and, as such, ensures software immutability except for authorized updates. However, CASU is similar to these $\mathcal{R}$A techniques with respect to updates.

Active RoTs proactively monitor $\mathcal{P}$rv behavior to prevent (or minimize the extent of) compromises. For example, [45–47] are architectures that guarantee execution of critical tasks even when all other software is compromised. Similarly, VERSA [48] guarantees sensor data privacy for low-end MCUs by allowing only authorized software to access and process sensed quantities. In contrast, CASU can be viewed as an active RoT that focuses on software immutability, prevention of illegal execution, and authorized updates.

Remote Over-the-Air (OTA) Updates support seamless delivery of software updates for IoT devices. Notably, TUF [49] is an update delivery framework resilient to key compromises. Uptane [50] extends TUF for supporting updates for vehicular ECUs. However, both TUF and Uptane require relatively heavy cryptographic operations, unsuitable for CASU-targeted low-end devices. ASSURED [13] extends TUF to provide a secure update framework for large-scale IoT deployments. SCUBA [51] uses software-based attestation to identify and patch infected software regions. However, due to the timing assumptions of software-based attestation, it is unsuitable for remote IoT settings. PoSE [52] and AONT [53] use proofs of secure erasure to wipe $\mathcal{P}$rv to show that its memory is fully erased and then install new software. However, these schemes are not fault-tolerant and can not retain previous software, in case of reset during erasure or new update installation. Also, an extensive discussion of various software update schemes can be found in [54].

Formal Verification provides increased confidence about the correctness of security techniques' implementations. In the space of low-end MCUs, *VRASED* [7] and *RATA* [14] are formally verified hybrid $\mathcal{R}$A architectures, where the latter one detects TOCTOU attacks. APEX [26] and PURE [11] offer formally verified proofs of remote software execution, and proof of update, reset, and erasure. Similarly, CASU offers a verified hardware module for authorized software immutability and unauthorized execution prevention.

8 CONCLUSIONS

In this paper, we designed CASU, a prevention-based root-of-trust architecture for low-end MCUs. CASU differs from prior work by disallowing illegal software modifications rather than detecting them. CASU also prevents execution of any unauthorized software and supports secure software updates. CASU is prototyped on OpenMSP430 and its hardware component is formally verified. Experiments show that CASU incurs quite low overhead and is thus suitable for resource-constrained low-end IoT devices. Its entire implementation is publicly available at [25].

Acknowledgments The authors sincerely thank ICCAD'22 reviewers. This work was supported by funding from NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs. The first author was supported in part by a seed grant from the ESL Global Cybersecurity Institute at RIT.

REFERENCES

[1] A. Francillon and C. Castellucia, "Code injection attacks on harvard-architecture devices," in *CCS '08*, 2008.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, 2013.

[3] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *IEEE DISCEX*, IEEE, 2000.

[4] OWASP, "Owasp top ten." https://owasp.org/www-project-top-ten/, 2021.

[5] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.

[6] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, 2017.

[7] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *USENIX Security*, 2019.

[8] M. Ammar, B. Crispo, and G. Tsudik, "Simple: A remote attestation approach for resource-constrained iot devices," in *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 247–258, IEEE, 2020.

[9] F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pp. 34:1–34:6, ACM, 2015.

[10] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[11] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," 2019.

[12] M. Ammar and B. Crispo, "Verify&revive: Secure detection and recovery of compromised low-end embedded devices," in *Annual Computer Security Applications Conference*, pp. 717–732, 2020.

[13] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[14] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," *CCS*, 2021.

[15] "Avr atmega 1284p 8-bit microcontroller." http://ww1.microchip.com/downloads/en/DeviceDoc/doc8059.pdf, 2009.

[16] T. Instruments, "Msp430 ultra-low-power sensing & measurement mcus." http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.

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

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

[19] O. Girard, "openMSP430," 2009.

[20] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS*, 2017.

[21] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, 1983.

[22] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSI Design*, 2004.

[23] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A tool for word-level verification," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[24] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, 2002.

[25] "CASU source code." https://github.com/sprout-uci/CASU, 2022.

[26] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*, (Boston, MA), USENIX Association, Aug. 2020.

[27] "Ultrasonic ranger code." https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger.

[28] "Temperature sensor code." https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor.

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

[30] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*, 2003.

[31] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Research in Security and Privacy (S&P)*, (Oakland, California, USA), pp. 272–282, IEEE, 2004.

[32] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SOSP*, 2005.

[33] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software attestation for key establishment in sensor networks," in *DCOSS*, 2008.

[34] R. W. Gardner, S. Garera, and A. D. Rubin, "Detecting code alteration by creating a temporary memory bottleneck," *IEEE TIFS*, 2009.

[35] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pp. 315–328, 2008.

[36] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1, pp. 13 – 22, 2008.

[37] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[38] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, eds.), pp. 743–754, ACM, 2016.

[39] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 24, ACM, 2017.

[40] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *Proceedings of the 36th International Conference on Computer-Aided Design*, pp. 384–391, IEEE Press, 2017.

[41] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1433–1449, IEEE, 2020.

[42] I. De Oliveria Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," in *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[43] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices," 2021.

[44] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, pp. 1–10, IEEE, 2019.

[45] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom, "Dominance as a new trusted computing primitive for the internet of things," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 1415–1430, IEEE, 2019.

[46] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado, "The lazarus effect: Healing compromised devices in the internet of small things," in *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020* (H. Sun, S. Shieh, G. Gu, and G. Ateniese, eds.), pp. 6–19, ACM, 2020.

[47] E. Aliaj, I. De Oliveira Nunes, and G. Tsudik, "GAROTA: generalized active root-of-trust architecture," *CoRR*, vol. abs/2102.07014, 2021.

[48] I. De Oliveira Nunes, S. Hwang, S. Jakkamsetti, and G. Tsudik, "Privacy-from-birth: Protecting sensed data from malicious sensors with VERSA," *CoRR*, vol. abs/2205.02963, 2022.

[49] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, p. 61–72, Association for Computing Machinery, 2010.

[50] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos, "Uptane: Securing software updates for automobiles," in *International Conference on Embedded Security in Car*, pp. 1–11, 2016.

[51] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *In Proceedings of the 5th ACM workshop on Wireless security (WiSe '06)*, p. 85–94, 2006.

[52] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure.," in *ESORICS*, 2010.

[53] G. O. Karame and W. Li, "Secure erasure and code update in legacy sensors," in *Trust and Trustworthy Computing*, pp. 283–299, Springer International Publishing, 2015.

[54] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, pp. 71907–71920, 2019.