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

Ivan De Oliveira Nunes
Rochester Institute of Technology

Youngil Kim
University of California Irvine

Sashidhar Jakkansetti
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 (RA), 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 RA, based on some unclear criteria. Thus, in case of prover’s compromise, verifier only learns about it late, upon the next RA 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 RA 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 RA in between software updates and leads to unobtrusive integrity assurance with guarantees akin to those of prior RA 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 (RA) [5–10], as well as proofs of remote software updates and memory erasure [11–13]. RA 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 RA by measuring a constant-size memory region that reflects the state 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, RA 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 RA 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 RA 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 RA.

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.
(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 ≈4 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 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 ≈4 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

RA, mentioned above, allows a trusted entity (verifier = Vrf) to remotely measure current memory contents (e.g., software) of an untrusted embedded device (prover = Prv). RA is usually realized as a simple challenge-manufacturing protocol:
(1) Vrf sends an RA request with a challenge (Chal) to Prv.
(2) Prv receives the request and computes an authenticating integrity check over its software memory region and Chal. The memory region can be either pre-defined or explicitly specified in the RA request.
(3) Prv returns the result to Vrf.
(4) Vrf verifies the result and decides if Prv is in a valid state. Although several RA 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 RA 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 RA technique which, though very fast, incurs a significant hardware cost over the baseline MCU.

VRASED[7] is a formally verified hybrid (hardware/software) RA 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 Prv memory (usually in PMEM). Meanwhile, its hardware component assures that its software counterpart executes securely and that no function of the RA secret key (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:

\[ H = \text{HMAC}(KDF(K, \text{Chal}), \text{AR}) \]  

where \( KDF(K, \text{Chal}) \) is a one-time key derived from the received Chal and \( K \) 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 RA techniques share a common limitation: they yield no information about the state of \( Prv \) software during the time between two consecutive RA instances. Consequently, it is impossible to detect the past presence of transient malware that: (1) infected \( Prv \), (2) remained active for a while, and (3) at some later time erased itself and restored \( Prv \) software to its “good” state. This holds as long as (1)-(3) occur between two successive RA 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
We consider an adversary, \textit{Adv}, that controls the entire memory state of \textit{Prv}, 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 \textit{Prv} software. It may also divert the execution control-flow to ignore update requests, as well as attempt to extract any \textit{Prv} secrets or forge update confirmations.

Furthermore, \textit{Adv} can configure DMA controllers on \textit{Prv} 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 \textit{Vrf} and \textit{Prv}, 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 \textit{Adv} physically reprograms \textit{Prv} 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].

### 3 CASU SCHEME & ASSUMPTIONS

#### 3.1 Basics

Similar to the typical RA setting, CASU involves a low-end MCU (\textit{Prv}) and verifier (\textit{Vrf}). The latter is a trusted higher-end device, e.g., a laptop, a smartphone, or a smart home gateway, or a device manufacturer’s back-end server. \textit{Vrf} 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 \textit{Vrf} for a given \textit{Prv}. Also, \textit{Prv} and \textit{Vrf} are assumed to share a master secret key (K) installed on \textit{Prv} 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 \textit{Prv} has sufficient computing capabilities\(^3\).

#### 3.2 Secure Update Overview

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

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

Upon receiving AAck, \textit{Vrf} executes the \textit{Verify}\(^{Vrf}\) procedure to check whether the AAck is a valid confirmation for the outstanding \textit{Update}\(^{Vrf}\). If no AAck is received, or if AAck verification fails, \textit{Vrf} assumes a failed update. Figure 2 illustrates the interaction between \textit{Vrf} and \textit{Prv}. Protocol details are described in Section 4 below.

#### 3.3 Adversary Model

We consider an adversary, \textit{Adv}, that controls the entire memory state of \textit{Prv}, 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 \textit{Prv} software. It may also divert the execution control-flow to ignore update requests, as well as attempt to extract any \textit{Prv} secrets or forge update confirmations.

\(^3\)In case of MSP430, based on our experimental attempts, neither generating nor even verifying public key signatures is viable.
CASU-HW: Hardware Security Monitor

CASU-HW monitors PC, Wen, Daddr, DMAen, DMAaddr 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:

\[
\text{Mod}_\text{Mem}(i) = (Wen \land Daddr = i) \lor (\text{DMAen} \land \text{DMAaddr} = i)
\]

\(i\) represents a memory address. \(\text{Mod}_\text{Mem}(i)\) 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 \(\text{Mod}_\text{Mem}(M)\). 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 \(\text{Mod}_\text{Mem}(M_1, M_2)\).

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., \(E_{\min}\) and \(E_{\max}\). 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 \(S_{\text{new}}\) to the same location and size of \(S_{\text{old}}\); 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, IVTR 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 Ctrl. Since ER is configurable via EP, after a secure update, CASU-SW re-configures EP to allow execution from the new ER location.
4.1.3 CASU-HW Properties Formally. Figure 5 formalizes the aforementioned CASU-HW security properties using propositional logic. Note that these properties must hold at all times. Equation 2 states that any modification to \( ER, EP, SF, IVTR \) when a program other than CASU-SW (PC \( \notin \) TCR) is executing – causes a \textit{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: \( \text{Update}^Vrf \), \( \text{Verify}^Vrf \) on \( Vrf \) and (\( \text{Auth}^Prv \), \( \text{Install}^Prv \)) on \( Prv \). At a high level, there are two ways of implementing it on \( Prv \). The CASU-HW properties are as follows:

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

(2) Download \( S\text{new} \) to PMEM (flash) and invoke \( \text{Auth}^Prv \). If \( \text{Auth}^Prv \) succeeds, \( \text{Install}^Prv \) updates \( EP \) to the location where \( S\text{new} \) resides. This is generally safer since \( S\text{new} \) and \( S\text{old} \) reside in two separate flash memory regions. If the installation is interrupted by a reset, CASU-SW can re-invoke \( \text{Install}^Prv \) to complete the installation. However, this requires \( \text{Prv} \) PMEM to be sufficiently large to accommodate both \( S\text{new} \) and \( S\text{old} \), 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 8KB, whereas the typical binary size is usually under 2KB.

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

4.2.1 \text{Update}^Vrf. Secure update requires for any software \( S\text{new} \) to be installed on \( Prv \) to adhere to the following format: \( S\text{new} := (L_{S\text{new}} || V_{S\text{new}} || N_{S\text{new}} || BINV_{S\text{new}} || IVTS_{S\text{new}}) \), where \( L_{S\text{new}} \), \( V_{S\text{new}} \), \( N_{S\text{new}} \) is the \( S\text{new} \) header consisting of its size, version number, and a random nonce, respectively. \( BINV_{S\text{new}} \) is the \( S\text{new} \) binary in byte-code that mandatorily includes a download and acknowledge subroutine that accepts future update requests and replies acknowledgment message back to \( Vrf \). \( IVTS_{S\text{new}} \) is the IVT of \( S\text{new} \) 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 \( V_{S\text{new}} \) should always be greater than the version number of the current (or old) software on \( Prv \). This avoids replay attacks that attempt to trick \( Prv \) into installing an old software version that contains vulnerabilities. In case \( Vrf \) wishes to revert to an older version (e.g., due to later-discovered bugs in \( S\text{new} \)), it must issue a brand new update request with the older-version software, though with a new \textit{version} number.

\( Vrf \), by invoking \( \text{Update}^Vrf \), computes \( \text{ATok} \) using equation 4 and sends \( (S\text{new}, \text{ATok}) \) to \( Prv \).

4.2.2 \text{Auth}^Prv. When \( Prv \) receives \( \text{Update}^Vrf \) with \( S\text{new} \) and \( \text{ATok} \), the current download subroutine on \( S\text{old} \) in \( ER \) accepts and downloads \( S\text{new} \) to an available PMEM slot. It then writes the pointers to \( S\text{new} \) to \( bEP \), buffer Executable Pointer, in PMEM, and writes \( \text{ATok} \) 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 \( \text{Auth}^Prv \), then \( \text{AAck} \) is not generated/sent, which is a clear indication to \( Vrf \) that the update was unsuccessful.

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

Note that CASU-SW execution is guarded by CASU-HW (which inherits \( \text{VRASED} \) 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 \( Vrf \) has to send a new update request. In this new request, \( Vrf \) 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 \text{Install}^Prv. Once \( S\text{new} \) is authenticated, \( \text{Install}^Prv \) is invoked. This is the critical step of Secure Update. It is responsible for updating the EP with \( bEP \), IVTR with \( IVTS_{S\text{new}} \) and computing authenticated acknowledgment \( \text{AAck} \) that is to be replied to \( Vrf \). 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 \( \text{AAck} \) failed, \( Vrf \) assumes that the update failed and repeats the update request with the same version number (since \( EP \) is updated to the new software), and \( \text{Auth}^Prv \) will fail again. Therefore, all three sub-steps must take place atomically. To this end, CASU-SW uses a \textit{Status flag} \( SF \) in PMEM, which it sets and unsets, before and after the completion of \( \text{Install}^Prv \) sub-steps, respectively.

To handle cases when a reset is triggered during \( \text{Install}^Prv \), 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 \( \{ \text{Update}^{\text{Vrf}}, \text{Auth}^{\text{Prv}}, \text{Install}^{\text{Prv}}, \text{Verify}^{\text{Vrf}} \} \) is realized as follows:

- \( K \) is a symmetric key pre-shared between \( \text{Vrf} \) and \( \text{Prv} \) (protected by VRASED secure architecture).

1. \( \text{Update}^{\text{Vrf}}(S_{\text{new}}) \rightarrow \text{ATok} \):
   
   \( \text{Vrf} \) generates a tuple \( T := (S_{\text{new}}, \text{ATok}) \), where \( S_{\text{new}} \) is the new software and \( \text{ATok} \) is the accompanying authentication token, as follows:
   
   (a) Compiles and generates \( S_{\text{new}} := (L_{\text{new}}, |V_{\text{new}}|N_{\text{new}}||B\text{IN}_{\text{new}}||IVT_{\text{new}}) \), where \( L_{\text{new}} \) is \( S_{\text{new}} \) size, \( V_{\text{new}} \) is \( S_{\text{new}} \) version number, \( N_{\text{new}} \) is a random nonce, \( B\text{IN}_{\text{new}} \) is \( S_{\text{new}} \) binary, and \( IVT_{\text{new}} \) is \( S_{\text{new}} \) IVT, to be placed in IVTR of \( \text{Prv} \).
   
   (b) Computes \( \text{ATok} \) using equation 4 with the second operand set to \( 0 || S_{\text{new}} \), where ‘0’ is the direction indicator from \( \text{Vrf} \) to \( \text{Prv} \).

   \[ \text{ATok} := \text{HMAC}(K, 0 || S_{\text{new}}) \]  

   (4)

2. \( \text{Auth}^{\text{Prv}}(S_{\text{new}}, \text{ATok}) \rightarrow \bot / \top \):
   
   Upon receiving a tuple \( T := (S_{\text{new}}, \text{ATok}) \) from \( \text{Vrf} \), \( S_{\text{new}} \) is downloaded at memory region pointed to by \( bEP \) and \( \text{ATok} \) is written to \( \text{ATR} \). Then \( \text{Prv} \) does the following:

   (a) If \( V_{\text{new}} \leq V_{\text{ER}} \), output \( \bot \) and return to \( \text{ER} \); otherwise, proceed to the next step.
   
   (b) Computes \( \sigma \) using equation 5.

   \[ \sigma := \text{HMAC}(K, 0 || bEP) \]  

   (5)

3. \( \text{Install}^{\text{Prv}}(S_{\text{new}}) \rightarrow \text{AAck} \):
   
   Upon invocation by \( \text{Auth}^{\text{Prv}} \), at boot time, in case \( \text{Status} \) is equal to 1, \( \text{Prv} \) does the following:

   (a) Sets \( \text{Status} \) to 1 and updates \( \text{EP} \) with values in \( bEP \).
   
   (b) Updates IVTR with \( IVT_{\text{new}} \).
   
   (c) Computes \( \text{AAck} \) using equation 6 and stores it at \( \text{ATR} \). In equation 6 the second operand is 1 \( |V_{\text{new}}|N_{\text{new}} \), where ‘1’ is the direction indicator from \( \text{Prv} \) to \( \text{Vrf} \).

   \[ \text{AAck} := \text{HMAC}(K, 1 || V_{\text{new}} | N_{\text{new}}) \]  

   (6)

   (d) Sets \( \text{Status} \) to 0 and jumps to new \( \text{ER} \), which is pointed to by the new value in \( \text{EP} \).

   \( \text{Prv} \) replies to \( \text{Vrf} \) with \( \text{AAck} \) indicating successful update.

4. \( \text{Verify}^{\text{Vrf}}(\text{AAck}) \rightarrow \bot / \top \):
   
   Upon receiving \( \text{AAck} \) from \( \text{Prv} \), \( \text{Vrf} \) does the following:

   (a) Computes \( \gamma \) using the same equation 6.
   
   (b) If \( \gamma \) == \( \text{AAck} \), outputs \( \top \); otherwise outputs \( \bot \).

There are other ways to mitigate the aforementioned \( \text{AAck} \) issues. Rather than storing \( \text{AAck} \) 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 \( \text{AAck} \) 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 \( \text{Vrf} \)-supplied timestamp instead of a nonce in \( S_{\text{new}} \) and modify \( \text{Auth}^{\text{Prv}} \) 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 \( \text{Vrf} \) issues a duplicate request. Each aforementioned alternative has its own benefits and drawbacks. We leave it up to \( \text{Vrf} \) to decide which is most suitable.

Note that none of the above can result in a DoS attack due to multiple requests, because all \( \text{Update}^{\text{Vrf}} \)s originate from a legit \( \text{Vrf} \) and are verified by \( \text{Auth}^{\text{Prv}} \). Moreover, download can check the \( S_{\text{new}} \) header to check if the request was already seen, discard the rest of the packets, and simply reply stored \( \text{AAck} \) to \( \text{Vrf} \).

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

Figure 6 depicts the workflow of secure updates. When \( \text{Prv} \) comes out of reset, it starts executing CASU-SW. CASU-SW first checks whether \( \text{Status} \) is 1, it invokes \( \text{Install}^{\text{Prv}} \) to resume installation of already verified \( S_{\text{new}} \) located at \( bEP \). Otherwise, it jumps
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 \( \text{reset} \). If \( \text{reset} \) is 1, the MCU core immediately resets.

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

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 \text{casu_entry}, \text{casu_authenticate}, \text{casu_install}, and \text{casu_exit}.

\text{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 \text{casu_entry}. \text{casu_entry} takes a boolean argument to test whether it was invoked at boot or by \text{ER} for an update. In the former case, it checks \text{Status} to determine whether to invoke \text{casu_install} in order to resume the unfinished update from the last reset. Otherwise, it calls \text{casu_exit}, which clears the MCU registers and jumps to the binary in \text{ER}. In the latter case, it invokes \text{casu_authenticate}. \text{casu_authenticate} checks for the validity of the version number of \text{S_new} at \text{bEP} and invokes \text{VRASED} software to compute HMAC. If the measurement matches \text{ATok}, \text{casu_install} is invoked; otherwise, it jumps to \text{casu_exit}. Finally, \text{casu_install} updates \text{EP}, copies the new IVT to \text{IVTR}, and computes and stores \text{AAck} at \text{ATR}. It also sets/unsets \text{Status} to indicate the status of installation to \text{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 \text{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 \text{VRASED}. Similar to prior work [5–7, 26], we consider additional Lookup Tables (LUTs) and registers. Compared to \text{VRASED}, CASU only requires 3% (99) additional LUTs and 0.3% (34) additional registers.

Verification Cost: CASU was verified using an 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 \text{VRASED} properties. The time to verify the composite design is under a second and requires 148MB of RAM.

| Architecture | Hardware LUTs | Regs | Location | 
|--------------|--------------|------|----------|
| OpenMSP430   | 1859         | 692  | -        | -        |
| VRASED       | 1902         | 724  | 481      | 10       | 0.4 s | 13.6 MB |
| CASU (+ VRASED) | 1958 | 726  | 576      | 12       | 0.9 s | 148 MB  |

Comparison with Related Architectures: In Figure 8, we compare CASU with other low-end MCU security architectures, including \text{VRASED} [7], \text{RATA} [14], \text{APEX} [26], and \text{PURE} [11], which provide RA-related services. However, recall that, unlike CASU, all these other architectures are reactive. As a superset of \text{VRASED},
they implement the following services: (1) memory integrity verification, i.e., \( RA \) [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 \( RA \) techniques with respect to updates.

Active RoTs proactively monitor \( Prv \) 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 \( Prv \) to show that its memory is fully erased and then install new software. However, these schemes are not fault-tolerant and cannot 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 \( RA \) 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.
