Wisecr: Secure Simultaneous Code Dissemination to Many Batteryless Computational RFID Devices

Yang Su, *Student Member, IEEE*, Michael Chesser, Yansong Gao, Alanson P. Sample, and Damith C. Ranasinghe

Abstract—Emerging ultra-low-power *tiny scale* computing devices run on harvested energy, are intermittently powered, have limited computational capability, and perform sensing and actuation functions under the control of a dedicated firmware operating without the supervisory control of an operating system. Wirelessly updating or patching firmware of such devices is inevitable. We consider the challenging problem of *simultaneous* and *secure* firmware updates or patching for a typical class of such devices—Computational Radio Frequency Identification (CRFID) devices. We propose *Wisecr*, the first secure and simultaneous wireless code dissemination mechanism to multiple devices that prevents *malicious code injection attacks* and *intellectual property (IP) theft*, whilst enabling *remote attestation of code installation*. Importantly, *Wisecr* is engineered to comply with existing ISO compliant communication protocol standards employed by CRFID devices and systems. We comprehensively evaluate *Wisecr*'s overhead, demonstrate its implementation over standards compliant protocols, analyze its security, implement an end-to-end realization with popular CRFID devices and open-source the complete software package on GitHub.

Index Terms—RFID, computational RFID, WISP, ISO 18000-63 Protocol, EPC protocol, secure wireless firmware update

1 Introduction

The maturation of energy-harvesting technology and ultra-low-power computing systems is leading to the advent of intermittently-powered, batteryless devices that operate entirely on energy extracted from the ambient environment [1]. The batteryless, low cost simplicity and the maintenance free perpetual operational life of these *tiny scale* computing platforms provide a compelling proposition for edge devices in the Internet of Things (IoT) and Cyber-Physical Systems.

Recent developments in *tiny scale* computing devices such as Wireless Identification and Sensing Platform (WISP) [2], MOO [3] and Farsens Pyros [4], or so called *Computational Radio Frequency Identification* (CRFID) devices, are highly resource limited, intermittently-powered, batteryless and operate on harvested RF (radio-frequency) energy. CRFID type devices are more preferable in scenarios that are challenging for traditional battery-powered sensors, for example, pacemaker control and implanted blood glucose monitoring [5] as well as domains where batteries are undesirable, for example, wearables for healthcare applications [6]. While, significant industrial applications are exemplified by asset management, such as monitoring and maintenance in the aviation sector [7], [8], [9], [10].

Despite various embodiments, the fundamental architecture of those devices include: microcontrollers, sensors, transceivers, and, at times actuators, with the most significant component, the *code* or software imparting the devices with the ability to communicate and realize interactive tasks [11], [12]. Consequently, the update and patching of this firmware is inevitable. In the absence of standard protocols or system level support, firmware is typically updated using a wired programming interface [3], [13]. In practical applications, the wired interface results in a potential attack vector to tamper with the behavior of the devices [14] and a compromised device can be further hijacked to attack other networked entities [15]. Disabling the wired interface after the initial programming phase at manufacture can prevent further access. But, leaves the wireless update option as the only method to alter the firmware or to re-purpose the devices post-manufacture.

However, *point-to-point communications protocols over limited bandwidth communication channels* characteristic of RFID systems poses problems for *rapid and secure* update of firmware over a wireless interface, post-manufacture. Notably, Federal Aviation Administration (FAA) in the United States has granted the installation of RFID tags and sensors on

- *Yang Su, Michael Chesser, and Damith C. Ranasinghe are with the Auto-ID Lab, School of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia. E-mail: {yang.su01, michael.chesser, damith.ranasinghe}@adelaide.edu.au.*
- *Yansong Gao is with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, Jiangsu 210094, China. E-mail: yansong.gao@njust.edu.cn.*
- *Alanson P. Sample is with the Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA. E-mail: apsample@umich.edu.*

Manuscript received 13 May 2021; revised 19 March 2022; accepted 20 April 2022. Date of publication 16 May 2022; date of current version 13 May 2023. This work was supported in part by Australian Research Council Discovery Program under Grant DP140103448, in part by the National Natural Science Foundation of China under Grant 62002167, and in part by the National Natural Science Foundation of JiangSu under Grant BK20200461. (Corresponding author: Damith C. Ranasinghe.) Digital Object Identifier no. 10.1109/TDSC.2022.3175313

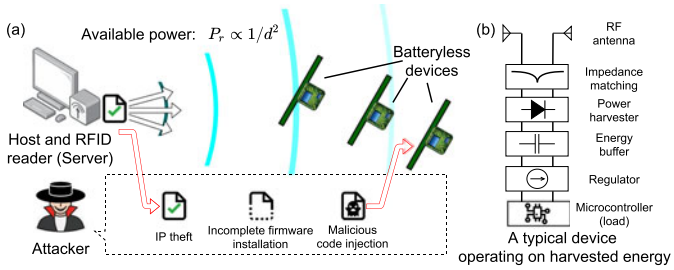

Fig. 1. (a) An attacker may: i) commit a malicious code injection attack such as alter code to inject bugs or load unauthorized code; ii) attempt to prevent a new firmware installation and spoof the acknowledgment signal of a successful update; and iii) commit IP theft by exploiting the insecure wireless channel. (b) A typical device architecture where the amount of power available for harvesting decreases exponentially with distance d from a powering source.

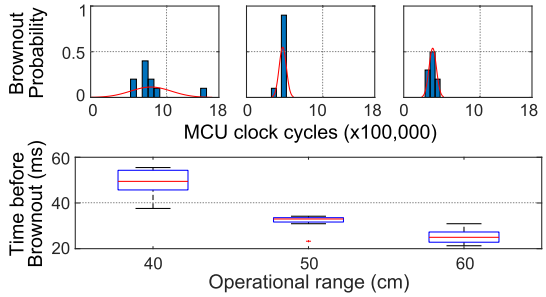

Fig. 2. The distribution of available clock cycles (top) and corresponding time (bottom) before *brownout*—exhaustion of available energy under computational load—for the open hardware and open software CRFID, WISP5.1LRG [13] built with a MSP430 microcontroller. We executed a message authentication code as the computational load and the device location is increased from 40 cm to 60 cm above a powering source—RFID reader antenna. We can observe that *available MCU clock cycles and operational time periods before state loss are limited per charging-brownout cycle*.

airplanes in 2018 [16], an increasing number of CRFID sensors are integrated with small aircraft and commercial airliners for maintenance history logging [7] and aircraft health monitoring [9]. In such a scenario, a high-efficiency (simultaneous) and secure firmware update is desirable to ensure operational readiness and flight safety.

The recent *Stork* [7] protocol addressed the challenging problem of fast wireless firmware updates to multiple CRFID devices. But, as illustrated in Fig. 1, *Stork* allows any party, authorized or not, armed with a simple RFID reader to: i) mount malicious code injection attacks; ii) attempt to spoof the acknowledgment signal of a successful update to fool the Server; and iii) steal intellectual property (IP) by simply eavesdropping on the over-the-air communication channel. Although a recent protocol [17] addressed the problem of malicious code injection attacks where by a single CRFID device is updated in turn, it neither supports simultaneous updates to many devices nor protects firmware IP and lacks a mechanism to validate the installation of code on a device.

Why Secure Wireless Firmware Update is Challenging?. Constructing a secure wireless firmware update mechanism for ultra-low power and batteryless devices is non-trivial; designing a secure method is challenging. We elaborate on the challenges below.

Limited and Indeterminate Powering. Energy-harvesting systems operate intermittently, only when energy is available from the environment. To operate, a device gradually buffers energy into a storage element (capacitor). Once sufficient energy is accumulated, the device begins operations. However, energy depletes more rapidly (e.g. milliseconds) during an operation compared to energy accumulation/charging (e.g. seconds). Further, energy accumulation in RF energy harvesting is sacrificed in backscatter communication links since a portion of the incident energy is reflected back during communications. Therefore, *power failures are common and occur in millisecond time scales* [7], [18], [19] as experimentally validated in Fig. 2.

Timing of power loss events across devices are ad-hoc. For example, harvested energy varies under differing distances; therefore, a computation task may only execute partially before power failure and it is hard to predict when it will occur.

Further, saving and subsequently retrieving state at code execution checkpoints for a long-run application via an intermittent execution model [20] is not only: i) costly in terms of computations and energy [21]—saving state in non-volatile memories such as Flash or Electrically Erasable Programmable Read-Only Memory (EEPROM) consumes more energy than static RAM (SRAM) whilst reading state from Ferroelectric Random Access Memory (FRAM) consumes more energy than writing as we demonstrate in Fig. 5; but also: ii) renders a device in a vulnerable state for an attacker to exploit when checkpoint state is stored on an off-chip memory, due to the lack of internal MCU memory, where non-invasive contact probes can be used to read-out contents when the memory bus used is easily accessible [22]. For example, common memory readout ports such as the I^2C (Inter-Integrated Circuit) bus used by a microcontroller for connecting to external memory devices are often exposed on the top layer of a printed circuit board (PCB). Hence, the content stored in an off-chip NVM can be easily read out using contact probes without any damage to the hardware [23].

These issues make the execution of long-run security algorithms, such as Elliptic Curve Diffie-Hellman (ECDH)[1] key exchange, difficult to deploy securely. Consequently, power must be carefully managed to avoid power loss and leaving the device in a potentially vulnerable state; and, we must seek computationally efficient security schemes.

Unavailability of Hardware Security Support. Highly resource-constraint devices lack hardware security support. Thus, security features, including a trusted execution environment (TEE)—for example, ARM Trustzone [26]—and dedicated memory to explicitly maintain the secrecy of a long term secret keys, as in [27], are unavailable.

Constrained Air Interface Protocols. The widely used wireless protocol for Ultra High Frequency (UHF) RFID communication only provides *insecure unicast communication links*

1. We are aware of optimized public key exchange implementations such as ECDH and Ring-LWE [24], [25]; however, they are impractical for passively powered resource-constrained devices. For example, ECDH [24] with Curve25519 still requires 4.88 million clock cycles on a MSP430 MCU. In contrast, there are a very limited number of clock cycles available from harvested-energy before energy depletion [17], [18]—e.g. 400,000 clock cycles expected even at close proximity of 50 cm from an energy source, see Fig. 2—and there is very limited time available for computations where a CRFID must reply before strict air interface protocol time-outs are breached.

and supports no broadcast features or device-to-device communication possible in mesh networking typical of wireless sensor networks.

Unavailability of Supervisory Control From An Operating System. Unlike *wireless sensor network nodes*, severely resource limited systems, such as CRFID, do not operate under the supervisory control of an operating system to provide security or installation support for a secure dissemination scheme.

1.1 Our Study

We consider the problem of secure and simultaneous code dissemination to multiple RF-powered CRFID devices operating under constrained protocols, device capability, and extreme on-device resource limitations—computing power, memory, and energy.

The scheme we developed overcomes the unique challenges, protects the firmware IP during the dissemination process, prevents malicious code injection attacks and enables remote attestation of code installation. More specifically, we address the following security threats:

- *Malicious code injection*: code alteration, loading unauthorized code, loading code onto an unauthorized device, and code downgrading.
- *Incomplete firmware installation*
- *IP theft*: reverse engineering from plaintext binaries.

Consequently, we fulfil the *urgent* and *unmet* security needs in the existing state-of-the-art multiple CRFID wireless dissemination protocol—Stork [7].

1.2 Our Contributions and Results

Contribution 1 (Section 2)—Wisecr is the first secure and simultaneous (fast) firmware dissemination scheme to multiple batteryless CRFID devices.

Wisecr provides three security functions for secure and fast updates: i) preventing malicious code injection attacks; ii) IP theft; and iii) attestation of code installation. *Wisecr* achieves *rapid* updates by supporting simultaneous update to multiple CRFID devices through a secure broadcasting of firmware over a *standard non-secure unicast air interface protocol*

Contribution 2 (Sections 3 & 4)—A holistic design trajectory, from a formal secure scheme design to an end-to-end implementation requiring only limited on-device resources.

Ultra-low power operating conditions and on-device resource limitations demand both a secure and an efficient scheme. First, we built an efficient broadcast session key exchange exploiting commonly available hardware acceleration for crypto on microcontroller units (MCUs).

Second, to avoid power loss and thus achieve uninterrupted execution of a firmware update session, we propose *new* methods: i) adaptive control of the execution model of devices using RF powering channel state information collected and reported by field deployed devices; ii) reducing disruptions to broadcast data synchronization across multiple devices by introducing the concept of a *pilot tag* selection from participating devices in the update scheme to drive the protocol. These methods avoid the need for costly, secure checkpointing methods and leaving a device in a vulnerable state during power loss.

Third, in the absence of an operating system, we develop an immutable bootloader to: i) supervise the control flow of the secure firmware update process; ii) minimize the occurrence of power loss during an update session whilst abandoning a session in case an unpreventable power loss still occurs; and iii) manage the secure storage of secrets by exploiting commonly available on-chip memory protection units (MPUs) to realize an immutable, bootloader-only accessible, secrets.

Contribution 3 (Section 4 and Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2022.3175313)—ISO Standards compliant end-to-end Wisecr implementation.

We develop *Wisecr* from specification, component design to architecture on the device, and implementation. We evaluate *Wisecr* extensively, including comparisons with current non-secure methods, and validate our scheme by an end-to-end implementation. *Wisecr* is a standard compliant secure firmware broadcast mechanism with a demonstrable implementation using the widely adopted air interface protocol—ISO-18000-63 protocol albeit the protocol's lack of support for broadcasting or multi-casting—and using commodity devices from vendors. Hence, the *Wisecr* scheme can be adopted in currently deployed systems. We demonstrate the firmware dissemination process here https://youtu.be/GgDHPJi3A5U.

Contribution 4—Open source code release. The tool sets and end-to-end implementation is open-sourced on GitHub: https://github.com/AdelaideAuto-IDLab/Wisecr.

Paper Organization. Section 2 presents the threat model, security requirements and *Wisecr* design; Section 3 details how the demanding security requirements are met under challenging settings; Section 4 discusses our end-to-end implementation, followed by performance and security evaluations; Section 5 discusses related work with which *Wisecr* is compared. Section 6 concludes this work.

2 *WISECR* DESIGN

In this section, we describe the threat model including a summary of notations in Table 1, followed by details of *Wisecr* design, and then proceed to identify the minimal hardware security requirements needed to implement the scheme.

2.1 Threat Model

Communication with an RFID device operating in the UHF range is governed by the widely adopted ISO-18000-63:2015—also known as the EPCglobal Class 1 Generation 2 version 2 (C1G2v2)—air interface protocol or simply the *EPC Gen2*.

Fig. 3 illustrates the communication channels in a networked RFID system. The RFID reader resides at the edge of the network and is typically connected to multiple antennas to power and communicate with RFID or CRFID devices energized by the antennas. The reader network interface is accessed by using a Low-Level Reader Protocol (LLRP) from a host machine. Communication with an RFID device operating in the UHF range is through the *EPC Gen2* protocol.

Our focus is on the insecure communication channel between the RFID reader connected antenna and the CRFID transponder or token $T_i \in \mathcal{T}$. Hence, we assume that the

TABLE 1
Table of Notations

$\mathcal{S}$	Server $\mathcal{S}$ is a single entity consisting of a host computer and a networked RFID reader.
$\mathcal{T}$	Tokens $\mathcal{T}$ is a set of individual (CRFID) devices T_i.
DB	Server's database, where each element is a three-tuple describing each CRFID token: i) the unique and immutable identification number $\mathbf{id}_i$; ii) the device specific secret key **k**; and iii) a flag denoting a device requiring an update **valid**.
firmware	The new firmware binary for the update.
$\mathbf{firmware}^{(j)}$	The j_{th} block of the **firmware**.
nver	The new version number (a monotonically increasing ordinal number with a one-to-one correspondence with each updated **firmware**).
ver	A token's current version number.
sk	The broadcast session key.
s	The MAC tag computed by the Server.
c, **r**	Attestation challenge and response, respectively.
$'$	An apostrophe denotes a value computed by a different entity, e.g., $\mathbf{s}'$ the MAC tag computed by a Token.
$_i$	A subscript i denotes a specific entity, e.g., $\mathbf{k}_i$ is the device key of the i_{th} Token.
$\langle\rangle$	Encrypted data, e.g. $\langle\mathbf{sk}_i\rangle$ denotes the encrypted session key **sk**, with the i_{th} token's key $\mathbf{k}_i$.
RNG()	A cryptographically secure random number generator.
SKP.Enc()	Symmetric Key Primitive encryption function described by $\langle\mathbf{m}\rangle \leftarrow \mathsf{SKP.Enc}_{\mathbf{sk}}(\mathbf{m})$. Here, the plaintext **m** is encrypted with the **sk** to produce the ciphertext $\langle\mathbf{m}\rangle$.
SKP.Dec()	Symmetric Key Primitive decryption function where $\mathbf{m} \leftarrow \mathsf{SKP.Dec}_{\mathbf{sk}}(\langle\mathbf{m}\rangle)$.
MAC()	Message Authentication Code function. By appending an authentication tag **s** to the message **m**, where $\mathbf{s} \leftarrow \mathsf{MAC}_{\mathbf{k}}(m)$, a message authentication code (MAC) function can verify the integrity and authenticity of the message by using the symmetric key **k**.
SNIFF()	The voltage $\mathbf{Vt}_i$ established by the power harvester within a fixed time t from boot-up at token i, is measured by SNIFF() described by $\mathbf{Vt}_i \leftarrow \mathsf{SNIFF}(t)$.
PAM()	Family of functions employed by the Power Aware Execution mode of operation proposed for the token to mitigate power-loss (brown-out) events.
`Query`	We denote *EPC Gen2* commands using typewriter fonts.

communication between a host and a reader is secured using standard cryptographic mechanisms [28]. Therefore, a host computer and a reader are considered as a single entity, the Server, denoted as $\mathcal{S}$ (a detailed execution of an *EPC Gen2* protocol session to *singulate* a single CRFID device from all visible devices in the field can be found in [7], [18]).

We assume a CRFID device can meet the pragmatic hardware security requirements, detailed in Section 2.3. Further, after device provisioning, the wired interface for programming is disabled—using a common technique adopted to secure resource-constrained microcontroller based devices [17], [19]. Subsequently, both the trusted party and adversary $\mathcal{A}$ must use the wireless interface for installing new firmware on a token.

Building upon relevant adversary models related to wireless firmware update for low-end embedded devices [29], [30], we assume an adversary $\mathcal{A}$ has full control over the communication channel between the Server $\mathcal{S}$ and the tokens $\mathcal{T}$. Hence, the adversary $\mathcal{A}$ can eavesdrop, manipulate, record and replay all messages sent between the Server $\mathcal{S}$ and the tokens $\mathcal{T}$. This type of attacker is referred to as an input/output attacker [31].

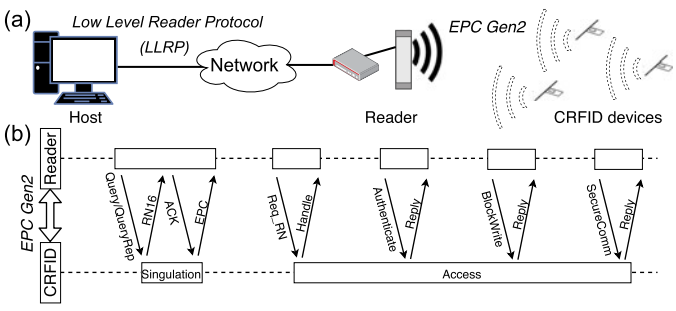

Fig. 3. (a) System overview. (b) Control/data flow over the *EPC Gen2* air interface. Firmware update will take place once a device enters the *Access* state and by implementing *Wisecr* over *Access* command specifications such as `Authenticate`.

We assume the firmware (application), potentially provided by a third party in the form of a stripped binary, may contain vulnerabilities or software bugs that can cause the program to deviate from the specified behavior with potential consequences being corruption of the bootloader and/or the non-volatile memory (NVM) contents. Such an occurrence is possible when firmware is frequently written in unsafe languages such as C or C++ [32], [33]. Hence, the firmware (application) cannot be trusted. In this context, similar to [29], [30], we also assume that the adversary cannot bypass any of the memory hardware protections (detailed in Section 2.3) and an adversary cannot mount invasive physical attacks to extract the on-chip non-volatile memory contents. Such an assumption is practical, especially in deeply embedded applications such as pacemaker control [18] where wireless update is the only practical mechanism by which to alter the firmware and physical access is extremely difficult.

As in [34], we also assume the adversary $\mathcal{A}$ cannot mount implementation attacks against the CRFID, or gain internal variables in registers, for example, using invasive attacks and side-channel analysis. We do not consider Denial of Service (DoS) attacks because it appears to be impossible to defend against such an attacker, for example, that disrupts or jams the wireless communication medium in practice [29].

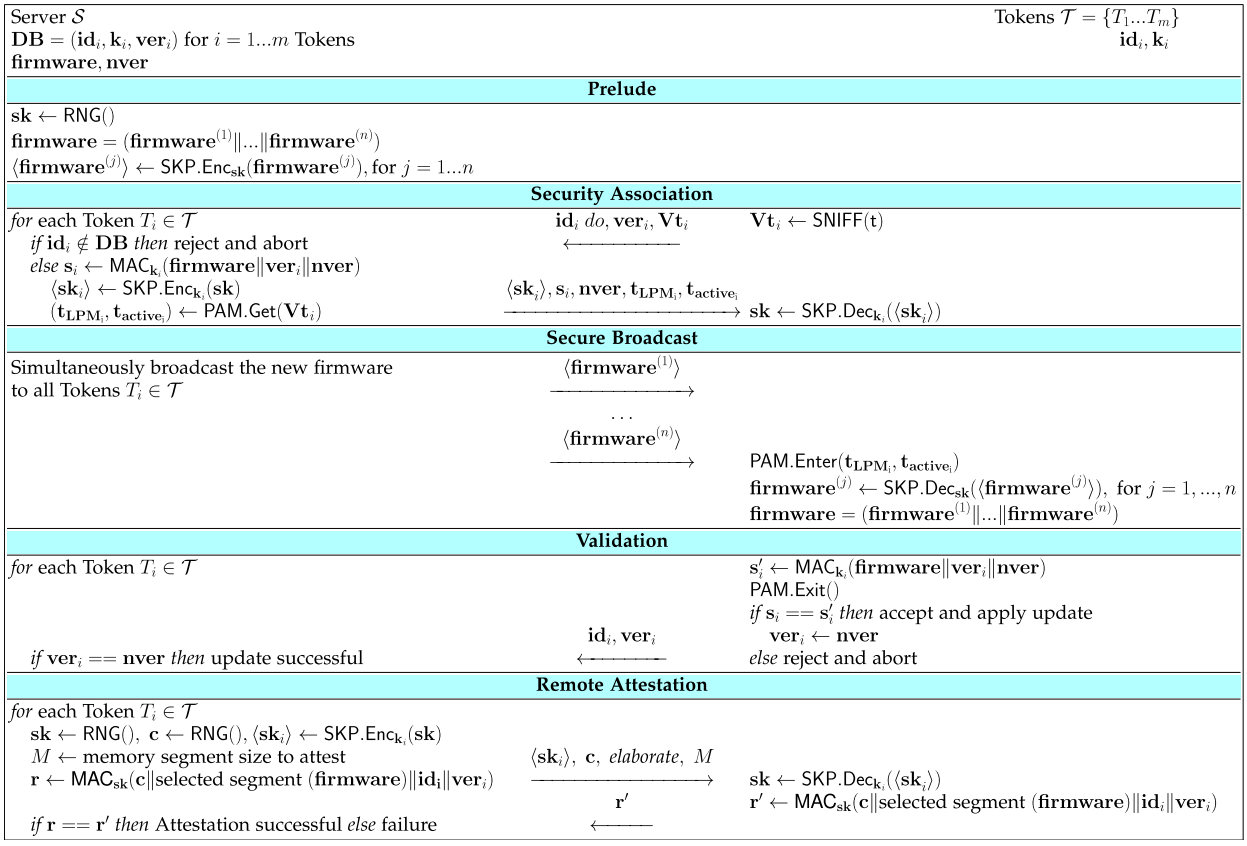

Fig. 4. *Wisecr*. The proposed wireless, secure and simultaneous code dissemination scheme to multiple tokens. Notably, the functions SNIFF() and PAM() facilitate the secure and uninterruptible execution of an update session. A single symmetric key cipher SKP can be exploited in practice to realise all of the primitives needed, including the MAC() function, to address the demands of a resource limited setting.

2.2 *Wisecr* Update Scheme

Wisecr enables the ability to securely distribute and update the firmware of multiple CRFID tokens, simultaneously. Given that a Server $\mathcal{S}$ must communicate with an RFID device $\mathcal{T}$ using *EPC Gen2*, *Wisecr* is compatible with *EPC Gen2* by design. Generally, our scheme can be implemented after the execution of the anti-collision algorithm in the media access control layer of the *EPC Gen2* protocol, where a reader must first singulate a CRFID device and obtain a handle, $RN16$, to address and communicate with each specific device. After singulating a device, the server can employ commands such as: `BlockWrite`, `Authenticate`, `SecureComm` and `TagPrivilege` specified in the *EPC Gen2* access command set [35], to implement the *Wisecr* scheme. We describe the update scheme in Fig. 4 and defer the details of our scheme implemented over the *EPC Gen2* protocol to the Appendix B, available in the online supplemental material.

As described in Table 1, the Server $\mathcal{S}$ maintains a database **DB** of provisioned tokens, issues the new *firmware* and the corresponding version number (**nver**).

Each token T_i has a secure storage area provisioned with: i) an $\mathbf{id}_i$, the immutable identification number; ii) $\mathbf{k}_i$, the device specific secret key stored in NVM that is read-only accessible by the immutable bootloader. The $\mathbf{k}_i$ assigned to different devices are assumed to be *independent and identically distributed* (i.i.d.); iii) $\mathbf{ver}_i$, the token's current firmware version number. The secure storage area is only accessible by the trusted and immutable bootloader provisioned on the device; this region is inaccessible to the firmware (application) and therefore cannot be modified by it.

We describe a dissemination session in four stages: i) Prelude; ii) Security Association; iii) Secure Broadcast; and iv) Validation. An update can be extended with an optional, v) Remote Attestation to verify the firmware installation.

STAGE 1: Prelude (Offline). In this stage, the Server $\mathcal{S}$ undertakes setup tasks. The Server uses RNG() to generate a broadcast session key (**sk**). The new *firmware* is divided into segments—or block; each block j is encrypted as $\langle \mathbf{firmware}^{(j)} \rangle \leftarrow \mathsf{SKP.Enc}_{\mathbf{sk}}(\mathbf{firmware}^{(j)})$, where $\langle \mathbf{firmware}^{(j)} \rangle$ denotes the encrypted firmware block j. The division of firmware is necessary as the narrow band communication channel and the *EPC Gen2* protocol does not allow *arbitrary* size payloads to be transmitted to a token.

STAGE 2: Security Association. In this stage, the Server $\mathcal{S}$ distributes the broadcast session key (**sk**) to all tokens $\mathcal{T}$ and builds a secure broadcast channel over which to simultaneously distribute the firmware to multiple tokens.

More specifically, each token in the energizing field of the Server responds with $\mathbf{id}_i$, $\mathbf{Vt}_i$ and $\mathbf{ver}_i$. The token will not be included in the following update session if: i) the $\mathbf{id}_i$ of the responding token is not in Server's **DB**; or ii) the token is not scheduled for an update ($\mathbf{valid}_i == \text{False}$). For tokens selected for an update, the Server computes a MAC tag $\mathbf{s}_i \leftarrow \mathsf{MAC}_{\mathbf{k}}\mathsf{i}(\mathbf{firmware}||\mathbf{ver}_i||\mathbf{nver})$. In practice, we cannot

assume that each token is executing the same version of the firmware, therefore a token specific MAC tag is generated over the device specific key whilst the firmware is encrypted with the broadcast session key.

The Server establishes a shared session key with each token T_i by sending $\langle \mathbf{sk}_i \rangle$, where $\langle \mathbf{sk}_i \rangle \leftarrow \mathsf{SKP.Enc}_{\mathbf{k}i}(\mathbf{sk})$ and $\mathbf{k}_i$ is specific to the i_{th} token, and **nver**. The $\langle \mathbf{sk}_i \rangle$ and $\mathbf{s}_i$ are transmitted to each token T_i. Each token decrypts the broadcast session key $\mathbf{sk} \leftarrow \mathsf{SKP.Dec}_{\mathbf{k}i}(\langle \mathbf{sk}_i \rangle)$—thus, all tokens selected for an update now possess the session key.

Notably, as detailed in Section 3.2.2, each token measures its *powering channel state* or its ability to harvest energy by measuring the voltage $\mathbf{Vt}_i$ established by the power harvester within a fixed time t from boot-up, as $\mathbf{Vt}_i \leftarrow \mathsf{SNIFF}(t)$ to transmit to the Server. There are two important reasons for measuring $\mathbf{Vt}_i$. First, to facilitate the power aware execution model (PAM) employed to mitigate power-loss at a given token. The Server uses the reported $\mathbf{Vt}_i$ to control the execution model of the i_{th} token. Specifically, the reported $\mathbf{Vt}_i$ is used by the Server to determine the length of time that a token dwells in low power mode (LPM) t_{LMP_i} and active mode t_{active_i} when executing computationally intensive tasks in the Secure Broadcast and Validation stages; here, the server determines $(t_{\text{LPM}_i}, t_{\text{active}_i}) \leftarrow \mathsf{PAM.Get}(\mathbf{Vt}_i)$. Second, the Server uses the reported $\mathbf{Vt}_i$ to realize the *Pilot-Observer Mode* of operation where one token is *elected* based on its $\mathbf{Vt}_i$, termed the *Pilot*, to control the flow in the Secure Broadcast stage by responding to server commands as detailed in Stage 3.

STAGE 3: Secure Broadcast. In this stage, the encrypted firmware blocks $\langle \mathbf{firmware}^{(1..n)} \rangle$ are broadcasted; and each token stores the new encrypted firmware blocks in its application memory region (Segment M defined in Section 3.1). Once the broadcast is completed, each token starts firmware decryption and validation. The $\langle \mathbf{firmware} \rangle$ is decrypted using the session key **sk** as $\mathbf{firmware}^{(j)} \leftarrow \mathsf{SKP.Dec}_{\mathbf{sk}}(\langle \mathbf{firmware}^{(j)} \rangle)$.

To realize a secure and power efficient logical broadcast channel under severely energy constrained settings, we use the *Pilot-Observer mode*. Herein, all tokens, except the *Pilot* token elected by the Server, enters into an *observer mode*. The tokens in the *observer mode* silently listen and store encrypted data disseminated by the server; the *Pilot* token performs the same operation whilst responding to the server commands. We employ two techniques within the Pilot-Observer Mode to mitigate power-loss and to achieve a secure broadcast to tokens: i) disabling energy consuming communication command reply from observers; and ii) the concept of electing a *Pilot* CRFID device to drive the update session as detailed in Section 3.2.2.

Notably, the techniques described in Stage 2 and 3 form the foundation for the uninterrupted execution of the bootloader to ensure security and enhance the performance of the firmware dissemination under potential power-loss events.

STAGE 4: Validation. In this stage, firmware is validated before installation. More precisely, a token specific MAC tag $\mathbf{s}'_i \leftarrow \mathsf{MAC}_{\mathbf{k}_i}(\mathbf{firmware}||\mathbf{ver}_i||\mathbf{nver})$ is computed by each token T_i. If the received MAC tag $\mathbf{s}_i$ matches the device computed $\mathbf{s}'_i$, the integrity of the firmware established and the issuing Server is authenticated by the token. Subsequently, the new firmware is updated and the new version number **nver** is stored as $\mathbf{ver}_i$. Otherwise, the firmware is discarded and the session is aborted. Notably, the *EPC Gen2* protocol provides a reliable transfer feature. Each broadcast payload is protected by a 16-bit Cyclic Redundancy Check (CRC-16) error detection method. Hence, the notification of a CRC failure to the Server results in the automatic re-transmission of the packet by the Server. Therefore, a MAC tag mismatch is more likely to be adversarial and discarding the firmware is a prudent action. At stage completion, each token switches from the *observer* or *Pilot* to the normal mode of operation after a software reset (reboot). Subsequently, all the temporary information such as session key **sk** and the token specific MAC tag $\mathbf{s}'_i$ in volatile memory will be erased.

Once the firmware is installed, the Server is acknowledged with the status of each participating token in the session. This is achieved by performing an *EPC Gen2* handshake after a reboot of the tokens, and comparing the version number $\mathbf{ver}_i$ reported from each token specified by $\mathbf{id}_i$ to the new firmware version number **nver** expected from the token. If the $\mathbf{ver}_i$ is up-to-date, the Server is acknowledged that the token T_i has been successfully updated.

Remark. It is theoretically possible to include a MAC tag in the acknowledgment message at the end of the Validation stage to authenticate the acknowledgment. But the implementation of this in practice is difficult under constrained protocols and limited resources typical of intermittently powered devices. This is indeed the case with the *EPC Gen2* air interface protocol and CRFID devices we employed. We discuss specific reasons in Appendix B, available in the online supplemental material, where we detail implementation of the *Wisecr* Update Scheme over the *EPC Gen2* air interface protocol.

STAGE 5 (Optional): Remote Attestation. The Server can elect to verify the firmware installation on a token by performing a remote attestation; a mechanism for the Server to verify the complete and correct software installation on a token. Considering the highly resource limited tokens, we propose a lightweight challenge response based mechanism re-using the $\mathsf{MAC}()$ function developed for *Wisecr*. The server sends a randomly generated challenge $c \leftarrow \mathsf{RNG}()$ and evaluates the corresponding response r' to validate the installation. We provision a new session key **sk** to enable the remote attestation to proceed independent of the previous stages, whilst avoiding the derivation of a key on device to reduce the overhead of the attestation routine.

We propose two modes of attestation; a *fast mode* and an *elaborate mode* to trade-off veracity of the verification against computational and power costs. The *fast* mode only examines the token serial number ($\mathbf{id}_i$) and the version number ($\mathbf{ver}_i$). While the *elaborate* mode traverses over an entire memory segment. The *elaborate* mode is relatively more time consuming but allows the direct verification of the code installed on the target token T_i.

We illustrate (in Fig. 4) and demonstrate (in Section 4.5) the *elaborate* mode (more veracious and computationally intensive) where response $\mathbf{r}' \leftarrow \mathsf{MAC}_{\mathbf{sk}}(\mathbf{c}||\text{selected segment}(\mathbf{firmware})||\mathbf{id}_i||\mathbf{ver}_i)$ attests the application memory segment containing the installed **firmware**. In contrast, the *fast* method computes the response as $\mathbf{r}' \leftarrow \mathsf{MAC}_{\mathbf{sk}}(\mathbf{c}||\mathbf{id}_i||\mathbf{ver}_i)$.

2.3 Token Security Requirements and Functional Blocks

Our design is intentionally minimal and requires the following security blocks.

Immutable Bootloader (Section 3.1) We require a static NVM sector M_{rx} that is write-protected to store the executable bootloader image to ensure the bootloader can be trusted post deployment where, for example, firmware (application) code vulnerabilities or software bugs do not lead to the corruption of the bootloader and the integrity of the bootloader can be maintained.

Secure Storage(Section 3.1) To store a device specific secret, e.g., $\mathbf{k}_i$, we require an NVM sector M read-*only* accessible by the immutable bootloader in sector M_{rx}. This ensures the integrity and security of non-volatile secrets post deployment since the firmware (application) code cannot be trusted, for example, due to potential vulnerabilities or software bugs that can lead to the corruption of non-volatile memory contents).

Uninterruptible Bootloader Execution (Section 3.2) During the execution of the bootloader stored in sector M_{rx}, execution cannot be interrupted until the control flow is intentionally released by the bootloader.

Efficient Security Primitives (Section 3.3) The update scheme requires: i) a symmetric key primitive; and ii) a keyed hash primitive for the message authentication code that are both computationally and power efficient.

In Section 3 we discuss how the associated functional blocks are engineered on typical RF-powered devices built with ultra-low power commodity MCUs.

3 On-device Security Function Engineering

To provide comprehensive evaluations and demonstrations, we selected the WISP5.1LRG [13] CRFID device with an open-hardware and software implementation for our token $\mathcal{T}$. This CRFID device uses the ultra-low power MCU MSP430FR5969 from Texas Instruments. Consequently, for a more concrete discussion, we will refer to the WISP5.1LRG CRFID and the MSP430FR5969 MCU in the following.

3.1 Immutable Bootloader & Secure Storage

For resource limited MCUs, several mechanisms—detailed in the Appendix A, available in the online supplemental material,—exists for implementing secure storage: i) Isolated segments; ii) Volatile keys; iii) Execute only memory; and iv) Runtime access protections. We opt for achieving secure storage and bootloader immutability using Runtime Access Protection by exploiting the MCU's memory protection unit (MPU), which offers flexibility to the bootloader. In particular, the MPU allows read/write/execute permissions to be defined individually for memory segments at power-up—prior to any firmware (application) code execution. *Wisecr* requires the following segment permissions to be defined by the bootloader to prevent their subsequent modifications through application code by locking the MPU:

Segment M is used as the secure storage area. During application execution, any access (reading/writing) to this segment results in an access violation, causing the device to restart in the bootloader.

Segment M_{rx} contains the bootloader, device interrupt vector table (IVT), shared code (e.g., *EPC Gen2* implementation). During application execution, writing to this segment results in an access violation.

Segment M_{rwx} covers the remaining memory, and is used for application IVT, code (**firmware**) and data.

3.2 Uninterruptible Bootloader Execution

The execution or control flow of the bootloader on the token must be uninterruptible by application code and power-loss events to meet our security objectives but dealing with brown-out induced power-loss events is more challenging. Power loss leaves devices in vulnerable states for attackers to exploit; therefore, we focus on innovative, pragmatic and low-overhead power-loss prevention methods. Our approach deliberately mitigates the chances of power-loss. In case a rare power-loss still occurs, the token will discard all state—including security parameters such as the broadcast session key; subsequently, the Server will re-attempt to update the firmware by re-commencing a fresh update session with this token. We detail our solution below.

3.2.1 Managing Application Layer Interruptions

The bootloader must be uninterruptible (by application code) for security considerations. For instance, the application code—due to an unintentional software bug or otherwise—could interrupt the bootloader while the device key is in a CPU register, so that the application code (exploited by an attacker) can copy the device key to a location under its control, or completely subvert access protections by overriding the MPU register before it is locked.

Recall, the memory segment M_{rx} (see Section 3.1) includes the memory region containing the IVT. This ensures that only the bootloader can modify the IVT. Since the IVT is under the bootloader's control, we can ensure that any non-maskable interrupt is unable to be directly configured by the application code, whereas all other interrupts are disabled during bootloader execution. Consequently, the interrupt configuration cannot be mutated by application code.

3.2.2 Managing Power-Loss Interruptions

Frequent and inevitable power loss during the bootloader execution will not only interrupt the execution, degrading code dissemination performance but also compromise security. Although intermittent computing techniques relying on saving and retrieving state at check-points from NVM—such as Flash or EPPROM—is possible, these methods impose additional energy consumption and introduce security vulnerabilities revealed recently [22].

Confronted with the complexity of designing and implementing an end-to-end scheme under extreme resource limitations, we propose an on-device Power Aware execution Model (PAM) to: i) avoid the overhead of intermittent computing techniques; and ii) enhance security without saving check-points to insecure NVM.

We observe that only a limited number of clock cycles are available for computations per charge and discharge cycle

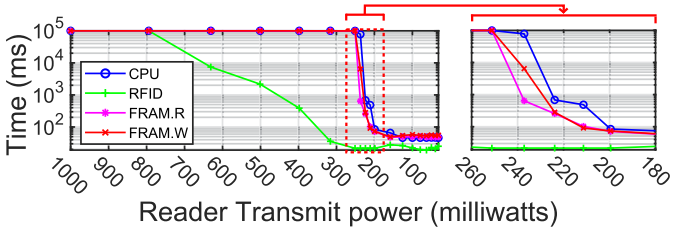

Fig. 5. Impact of four key device tasks on power-loss: CPU (computations); FRAM.R/W (memory read/write accesses); and RFID (communications). The data above are collected by using special firmware. The power-loss event is captured by monitoring through a GPIO pin. We conduct 10 repeated measurements and report the mean time before power-loss. The plot provides a lateral comparison among four operations. The data is measured from a single CRFID device with oscilloscope probes attached to measure the device's internal state.

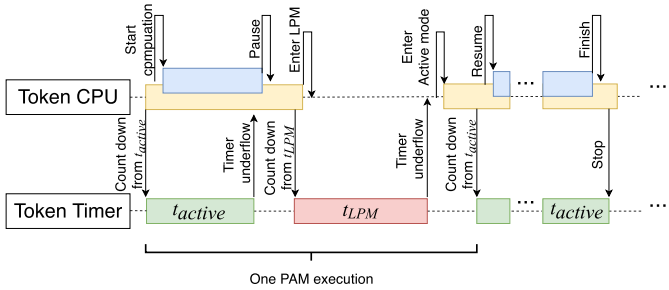

Fig. 6. A sequence diagram describing our proposed PAM and illustrating the interaction between the Token's CPU execution and the hardware timer. One PAM execution cycle consists of an active mode followed by a low power mode (LPM), one complete task may involve multiple PAM cycles. Other operations, such as RFID communications and FRAM access, are coordinated by the token CPU. If the CPU is in the LPM state, the entire system will be halted to allow energy to accumulate.

(Intermittent Power Cycle or IPC) of a power harvester, as illustrated via comprehensive measurements in Fig. 2. Further, the rate of energy consumption/depletion is faster than energy harvesting. We recognize that there are three main sources of power-loss: i) (CPU) energy required for function computation exceeding the energy supply capability from the harvester; ii) (FRAM.R/W) memory read/write access such as in executing `Blockwrite` commands; and iii) (RFID) power harvesting disruptions from communications—especially for backscattering data in response to *EPC Gen2* commands.

To understand the severity of these four causes, we measure the maximum time duration before brownout/power-loss versus the harvested power level for each task—CPU, FRAM.R, FRAM.W and RFID. In the absence of a controlled RF environment (i.e., anechoic chamber), it is extremely difficult to maintain the same multipath reflection pattern. Especially when changing the distance between the radiating reader antenna and an instrumented CRFID device considering the multipath interference created by the probes, cables, and the nearby oscilloscope and researcher to monitor the device's internal state. To minimize the difficulty of conducting the experiments, we place the CRFID device at a fixed distance (20 cm) whilst keeping all of the equipment at fixed positions, and adjust the transmit power of the RFID reader through the software interface. According to the free-space path loss equation [36], adjusting the transmit power of the RFID reader or changing the distance can be used to vary the available power at the CRFID device. We describe the detailed experimental settings in Appendix H, available in the online supplemental material. For experiments without the requirement for monitoring the device's internal state, we still employ distance-based measurements as in previous studies [7], [17], [21].

The results are detailed in Fig. 5. For a transmit power greater than 800 mW, the CRFID transponder continuously operates without power failure within 100 seconds. If the reader transmit power is below 800 mW, the average operating time of the RFID task drops as the power level decreases. This is because the RFID communication process invokes shorting the antenna, during which the energy harvesting is interrupted. Notably, different from Flash memory, reading data from FRAM (FRAM.R) consumes more power than writing to it (FRAM.W) as a consequence of the destructive read and the compulsory write-back [37]. Consequently, we developed:

- The *Pilot-Observer Mode* to reduce the occurrence of RFID tasks by enabling observing devices to listen to broadcast packets in silence whilst *electing* a single *Pilot* token to respond to the Server.
- The *Power Aware Execution Model (PAM)* to ensure memory access (FRAM tasks) and intensive computational blocks of the security protocol (CPU tasks) do not exceed the powering capability of the device.

Power Aware Execution Model (PAM). The execution mode enables a token to dynamically switch between *active* power mode and lower power mode (LPM)—LPM preserves (SRAM) state and avoids power-loss while executing a task. PAM is illustrated in Fig. 6. In active power mode, the token executes computations, and switches to LPM before power-loss to accumulate energy; subsequently, the token is awoken to active power mode to continue the previous computation after a period of t_{LPM}.

Our PAM model builds upon [17], [38] in that, these are designed for execution scheduling to *prevent power-loss from brownouts*. Compared to [17], we consider dynamic scheduling of tasks and in contrast to [38] sampling of the harvester voltage (*only possible in specific devices*) within the application code, we consider dynamic scheduling determined by the more resourceful Server $\mathcal{S}$ using a single voltage measurement reported by a token $\mathcal{T}$ (see Appendix C, available in the online supplemental material, for detailed comparison). We outline the means of achieving our PAM model on a token below.

During the Security Association stage shown in Fig. 4, a token measures and reports the voltage $\mathbf{Vt}_i \leftarrow \text{SNIFF}(t)$ to the Server. The $\mathbf{Vt}_i$ measurement indicates energy that can be harvested by token T_i under the settings of the current firmware update session. According to $\mathbf{Vt}_i$, the Server determines the active time period t_{active} and LPM time period t_{LPM} for each CRFID device (detailed development of a model to estimate t_{active} and t_{LPM} from $\mathbf{Vt}$ is in Appendix D, available in the online supplemental material). *Consequently, each CRFID device's execution model is configured by the Server with device specific LPM and active periods at run-time. Hence, an adaptive execution model customized to the available power that could be harvested by each CRFID device is realized. Notably, this execution scheduling task is outsourced to the resourceful Server.*

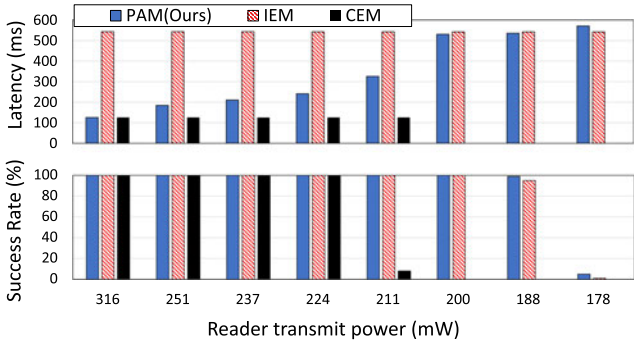

Fig. 7. Experimental evaluation of the latency and success rate of our proposed power aware execution model (PAM) compared to the Intermittent Execution Mode (IEM) method in [17] and a typical continuous execution model (CEM). The long-run task used in this evaluation is a MAC computation over a 1,536-Byte random message typically requiring 125.5 ms to complete in CEM. Notably, CEM fails when the reader transmit power is below 200 mW. We conduct 100 repeated measurements and report the mean.

In this context, we assume the distances between the antenna and target CRFID devices are relatively constant during the *short* duration of a firmware update. Firmware updates are generally a maintenance activity where CRFID integrated components are less likely to be mobile to ease maintenance, such as during the scheduled maintenance of an automated production line [39]; night time updates in smart buildings when people are less likely to be present and facilities are inoperative [40]; or the pre-flight maintenance or inspection of aircraft parts while parked on an apron [9]. Further, it is desirable to maintain a stable powering channel in practice by ensuring a consistent distance during the maintenance or patching of devices for a short period. This is a more reasonable proposition than the wired programming of each device. Hence, the relatively fixed distance is a reasonable assumption in practice. Notably, given the challenging nature of the problem, previous non-secure firmware update methods, such as R^3 [21] and *Stork* [7], were evaluated under the same assumption.

PAM Experimental Validation. To understand the effectiveness of our PAM method to reduce the impact of brown-out, we execute a computation intensive module, MAC() using PAM, at power-up on a CRFID. We employ a few lines of code to toggle GPIO pins to indicate the successful completion of a routine. Notably, it is difficult to track a device's internal state without a debug tool attached to the device; however, if the debug interface is in use, it will either interfere with powering, or affect the timing by involving additional Joint Test Action Group (JTAG) service code. We measure the time taken to complete the MAC() execution (latency) and success rate (success over 10 repeated attempts under wireless powering conditions) with digital storage oscilloscope connected to GPIO pins indicating a successful execution before power loss. We compute a MAC over a 1,536-Byte randomly generated message to test the effectiveness of PAM in preventing a power loss event due to brown-out and compare performance with the execution method—denoted IEM—of a fixed LPM state (30 ms) *programmatically* encoded during the device provisioning phase as in [17].

The results in Fig. 7 show the effectiveness of PAM to mitigate the interruptions from power loss. This is evident when the success rate results without PAM—using the continuous execution model (CEM)—is compared with those of PAM at decreasing operating power levels. However, as expected, we can also observe that the dynamically adjusted execution model parameters (t_{LPM} and t_{active}) of PAM at decreasing power levels to prevent power loss events and increase latency or the time to complete the routine. When PAM is compared with IEM, the dynamically adjusted execution model parameters allow PAM to demonstrate an improved capability to manage interruptions from power-loss at poor powering conditions; this is demonstrated by the higher success rates when the reader transmit power is at 188 mW and 178 mW. Since IEM encodes operating settings programmatically during the device provisioning phase [17], we can observe increased latency at better powering conditions when compared to PAM which allows a completion time similar to that obtained from CEM when power is ample. Thus, PAM provides a suitable compromise between latency and successful completion of a task at different powering conditions.

Notably, the RFID media access control (MAC) layer on a CRFID device is implemented in software as assembly code and executed at specific clock speeds to ensure strict signal timing requirements in the *EPC Gen2* air interface protocol. Additionally, protocol message timing requirements places strict limits on waiting periods for devices responses. Hence, we do not consider the direct control of the execution mode during communication sessions for managing power consumption and instead rely on the Pilot-Observer mode method we investigate next.

Pilot-Observer Mode.

We observed and also confirmed in Fig. 5 that the task of responding to communication commands will likely cause power loss. During the Secure Broadcast stage shown in Fig. 4, communication is dominated by repeated `SecureComm` commands with payloads of encrypted **firmware** and the data flow is uni-directional. Intuitively, we can disable responses from all the tokens to save energy. However, the `SecureComm` command under *EPC Gen2* requires a reply (ACK) from the token to serve as an acknowledgment [35]. An absent ACK within 20 ms will cause a protocol timeout and execution failure.

To address the issue, we propose the pilot-observer mode inspired by the method in Stork [7]. A critical and distinguishing feature of our approach is *the intelligent election of a pilot token from all tokens to be updated*—we defer to Appendix E, available in the online supplemental material, for a more detailed discussion on the differences. As illustrated in Fig. 8a, our approach places all in-field tokens to be updated into an *observer* mode except one token *elected by the Server* to drive the *EPC Gen2* protocol—this device is termed the *Pilot* token. By doing so, observer tokens process all commands such as `SecureComm`—ignoring the handle designating the target device for the command—whilst remaining silent or muting replies to *all* commands whilst in the *observer* mode. Muting replies significantly reduce energy consumption and disruption to power harvesting of the tokens.

We propose *electing* the token with the *lowest* reported **Vt** as the pilot based on the observation: measuring powering channel state from the token, obtained from $\mathbf{Vt}_i \leftarrow \text{SNIFF}(t)$,

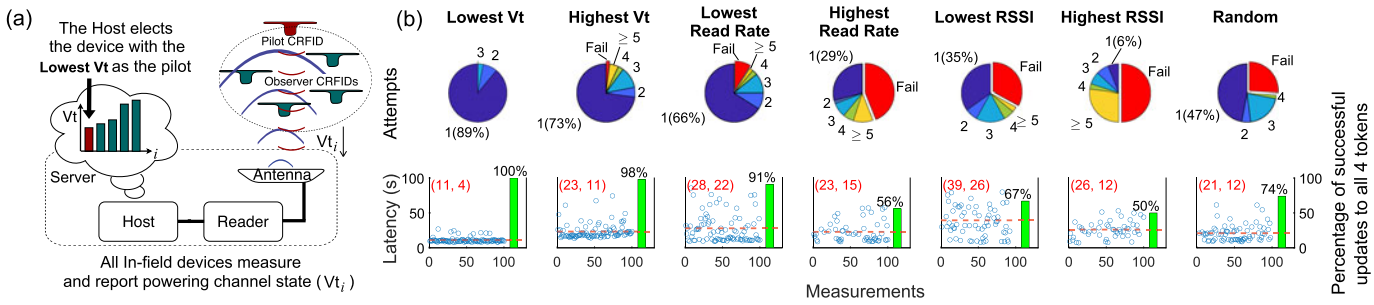

Fig. 8. (a) Our proposed Pilot-Observer method. Only the elected *Pilot* using the Vt_i based election method responds while *observers* listen silently. (b) Evaluation of Pilot Token selection strategies. Measured Tokens are placed 40 cm above the reader antenna, where the powering condition is critical; we defer the reader to Appendix F, available in the online supplemental material, for further results from other operational distances. The number of attempts to have all 4 CRFIDs (Tokens) updated (pie chart) and the corresponding latency (scatter plot) over 100 repeated measurements. The mean number and the standard division for successful updates for all four tokens are also labeled in the scatter plot with (mean, std) given in seconds.

is the most reliable measure of power available to a given device (see the discussion in Appendix D, available in the online supplemental material).

We recognize that it is very difficult to implement an explicit synchronization method to ensure the observer tokens stay synchronized with the pilot token in the secure broadcast update session. This is due to the level of complexity and overhead that an explicit method would bring and the consequence of such an overhead would be the increased occurrence of failures of a secure broadcast update session due to the additional task demands on tokens—we discuss the problem further in Section 6. Although synchronization between the pilot and observer tokens are not explicit, the Pilot-Observer method implicitly enforces a degree of synchronicity. The selected pilot token—tasked with responding to the Host commands during the broadcast—has to spend more time than observer tokens to prepare the uplink packet and send a reply (ACK) to the Host (RFID reader) to meet the *EPC Gen2* specification requirements as summarized in Table 2; hence the update process is controlled by the *slowest* token. Further, since we elect the token with the lowest powering condition as the pilot, the update is controlled by the most energy-starved token. This strategy leads to other tokens having higher levels of available power and extra time to successfully process the broadcasted firmware and remain synchronized with the update process. Hence, *if the pilot succeeds, observers are expected to succeed*.

Validation of Pilot Election Method. To demonstrate the effectiveness of our pilot token election method, we considered ***seven*** different pilot token selection methods based on token based estimations of available power and indirect methods of estimating the power available to a token by the server: 1) Lowest V_t: this implies the pilot is selected based on the most conservative energy availability at token; 2) Highest V_t: the pilot has to reply to commands, hence we may expect higher power availability to prevent the failure of a broadcast session due to brownout at the pilot; 3) Lowest Read Rate: a slow pilot allows observers to gather more energy during the broadcast and remain synchronized with the protocol to prevent failure of the observer tokens; 4) Highest Read Rate: a faster pilot could reduce latency; 5) Lowest Received Signal Strength Indicator (RSSI): successful communication over the poorest channel may ensure all observers are on a better channel (RSSI acts as an indirect measure of the powering channel at a token); 6) Highest RSSI: prevent pilot failure due to a potentially poor powering channel at the pilot token; and 7) Random selection: *monkey beats man on stock picks*.

We have extensively evaluated and compared all of the above seven pilot token selection methods. The experiment is conducted by placing 4 CRFID tokens at 20 cm, 30 cm, 40 cm and 50 cm above an reader antenna; each CRFID is placed in alignment with the four edges of the antenna (See Fig. 12d). We repeated the code dissemination process *100 times* at each distance test, for each of the 7 different pilot selection methods. We recorded two measures: i) latency (seconds); and ii) number of times the broadcast attempt updated all of the 4 in-field devices.

As illustrated in Fig. 8b, at 40 cm where the power conditions are more critical at a token, the selected pilot token with the lowest token voltage (**Vt**) shows an enormous advantage, in terms of both latency and attempt number. Overall, our proposed approach (electing the lowest **Vt**) ensures that most observer tokens are able to correctly obtain and validate the firmware in a given broadcast

TABLE 2
Execution Overhead of Pilot and Observer Tokens When Receiving Broadcast Packets

	Clock Cycles[1]	Memory Usage (Bytes)		Operation
		ROM	RAM	
Pilot token receiving	23,082	12	2	apply decoding, prepare reply (e.g., compute CRC)
Pilot token replying	1,131	0	0	communication (backscattering)
Observer token receiving	22,002	12	2	apply decoding

1 *Numbers are collected using a JTAG debugger where the reported values are averages over 100 repeated measurements. We provide a detailed discussion of the experiments and analysis process in Appendix G, available in the online supplemental material.*

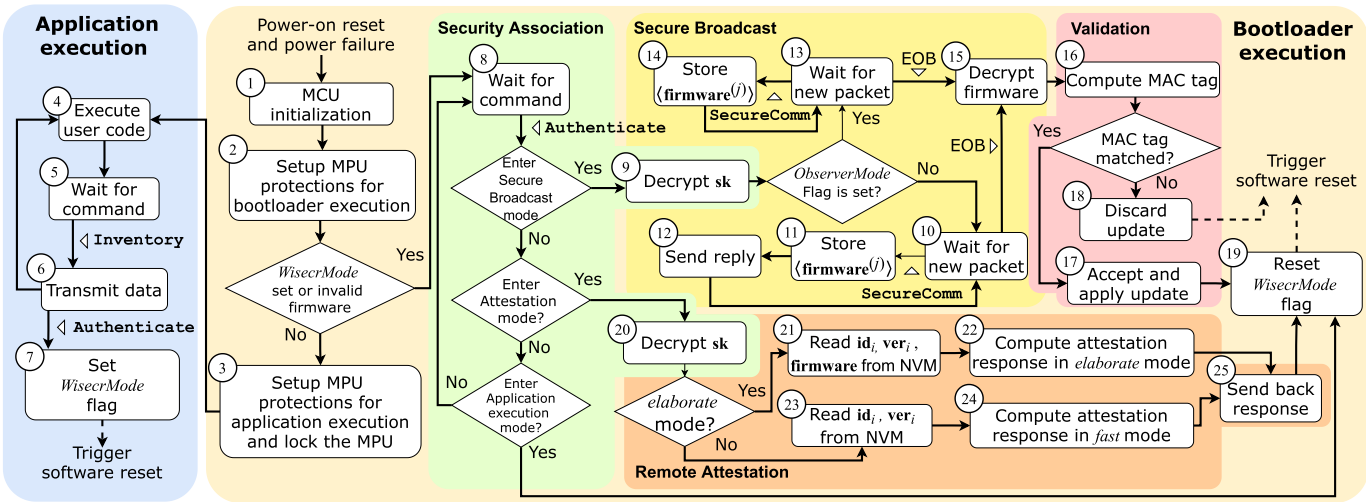

Fig. 9. *Wisecr* Bootloader control flow. The Bootloader manages Security Association, Secure Broadcast, Validation and Remote Attestation stages.

session on the first attempt in the critical powering regions of operation (we provide a theoretical justification in Appendix D, available in the online supplemental material).

We also evaluated the reduction in power consumption achieved from computational tasks, in addition to eliminating the communication task. While we provide a detailed discussion of the experiments and analysis process in Appendix G, available in the online supplemental material, we summarize our experimental results in Table 2. The measurements show that the observer tokens, compared to the pilot token, consumes 2,211 less clock cycles *per* firmware packet from the Host. This is a reduction of 9.13% to process each firmware packet sent from the Host.

Summary. Our pilot-election based method effectively reduces the chance of power failure as well as de-synchronization of the observer tokens during a firmware broadcast session. Our approach is able to ensure that more of the observer tokens are able to correctly obtain and validate the firmware during a single broadcast sessions.

3.3 Efficient Security Primitives

Wisecr requires two cryptogrpahic primitives: i) symmetric key primitive SKP(); and ii) a keyed hash primitive for the message authentication code MAC(). When selecting corresponding primitives in the following, two key factors are necessary to consider:

1) *Computational cost*: The CPU clock cycles required for a primitives quantifies both the computation cost and energy consumption. Therefore, clock cycles required for executing the primitives need to be within energy and computational limits of the energy harvesting device.
2) *Memory needs*: On-chip memory is limited—only 2 KiB of SRAM on the target MCU—and must be shared with: i) the RFID protocol implementation; and ii) user code.

Symmetric Key Primitives. We fist compare block ciphers via software implementation benchmarks—clock cycle counts and memory usage—on our target microcontroller [41], [42]. We also considered the increasingly available cryptographic co-processors in microcontrollers: our targeted device uses an MSP430FR5969 microcontroller and embeds an on-chip *hardware* Advanced Encryption Standard (AES) accelerator (we refer to as HW-AES) [43]. Based on the above selection considerations, HW-AES is confirmed to outperform others. Specifically, HW-AES to encrypt/decrypt a 128-bit block consumes 167/214 clock cycles with a power overhead of 21 μA/MHz; therefore we opted for HW-AES for SKP.Dec function implementation on our target device. We configured AES to employ the Cipher-Block Chaining (CBC) mode[2] similar hardware AES resources can be found in a variety of microcontrollers, ASIC, FPGA IP core and smart cards. In the absence of HW-AES, a software AES implementation, such as tiny-AES-c can be an alternative.

Message Authentication Code. MACs built upon BLAKE2s-256, BLAKE2s-128, HWAES-GMAC and HWAES-CMAC on our targeted MSP430FR5969 MCU were taken into consideration [17]. We selected the 128-bit *HWAES-CMAC*—Cipher-based Message Authentication Code[3]—based on AES since it yielded the lowest clock cycles per Byte by exploiting the MCU's AES accelerator (HW-AES).

3.4 Bootloader Control Flow

We can now realize an uninterruptible control flow for an immutable bootloader built upon on the security properties identified and engineered to achieve our *Wisecr* scheme described in Section 2.2. We describe the bootloader control flow in Fig. 9.

At power-up, ① the token performs MCU initialization routines, ② setup MPU protections for bootloader execution and carries out a self-test to determine whether a firmware update is required, or if there is a valid application installed. If no update is required and the user application is valid, ③ the bootloader configures MPU protections for application

2. We are aware of CBC padding oracle attacks; however, in our implementation, the response an attacker can obtains is to the CMAC failure or success and not the success or failure of the decryption routine. Alternatively, the routines can be changed to employ authenticated encryption in the future, with an increase in overhead.

3. The implementation in NIST Special Publication 800-38B, *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication* is used here.

execution, before handing over control to the application. During the application execution, ④ the user code is executed, ⑤ then the token T waits for command from the Server $\mathcal{S}$, the Server $\mathcal{S}$ may send an `Inventory` commands to instruct the token T to ⑥ send back e.g., sensed data, or in ⑦ setup the *WisecrMode* flag and trigger a software reset in preparation for an update.

Security Association. Upon a software reset, a set *WisecrMode* flag directs the token T to enter the Security Association stage. ⑧ Subsequently, the token waits for further instructions from the Server. On reception of an `Authenticate` command, carrying an encrypted broadcast session key $\langle \mathbf{sk} \rangle_i$ and the MAC tag $\mathbf{s}_i$ of the new firmware, ⑨ the token T decrypts $\langle \mathbf{sk} \rangle_i$ with its device key $\mathbf{k}_i$ and acquires the session key $\mathbf{sk}$.

Secure Broadcast. Recall, at this stage, all tokens selected for update will be switched into the observer mode except the pilot token that is set to respond to the Server. The pilot token in state ⑩ receives a new chunk of encrypted firmware $\langle \mathbf{firmware}^{(j)} \rangle$, and ⑪ stores it into the specified memory location. ⑫ The pilot token sends reply to the Server. In ⑬, for tokens in observer mode, they silently listen to the communication traffic between the Server and the Pilot token without replying. In other words, tokens under observer mode receives the encrypted firmware chunks, ⑭ store chunks in memory but ignores unicast handle identifying the target device; this significantly saves observers' energy from replying as detailed in Section 3.2.2. Once an End of Broadcast (`EOB`) message is received, all tokens stop waiting for new packets and start firmware decryption ⑮.

Validation. The token computes a local MAC tag $\mathbf{s}'_i$, including the decrypted firmware, and compares it with the received MAC tag $\mathbf{s}_i$ ⑯. The firmware is accepted, applied to update ⑰ the token, if $\mathbf{s}'_i$ and $\mathbf{s}_i$ are matched, and ⑲ the *WisecrMode* flag is reset; otherwise, ⑱ the firmware is discarded and the update is aborted. Subsequently, each token performs a software reset to execute the new firmware or reinitialize in bootloader mode if the firmware update is unsuccessful.

Remote Attestation. On reception of an `Authenticate` command with an instruction to perform remote attestation, ⑳ the token first decrypts the session key $\mathbf{sk}$ and reads ㉑ $\mathbf{id}_i$, $\mathbf{ver}_i$ and $\mathbf{firmware}$ from the NVM according to the specified memory address and size (firmware is only used in the *elaborate* mode, in the *fast* mode ㉓ only $\mathbf{id}_i$ and $\mathbf{ver}_i$ are involved). The attestation response is then computed as $\mathbf{r} \leftarrow \mathsf{MAC}_{\mathbf{sk}}(\mathbf{c} \| \mathbf{selected\ segment}\ (\mathbf{firmware}) \| \mathbf{id_i} \| \mathbf{ver}_i)$ if the *elaborate* mode is selected ㉒ or $\mathbf{r} \leftarrow \mathsf{MAC}_{\mathbf{sk}}(\mathbf{c} \| \mathbf{id_i} \| \mathbf{ver}_i)$ if using the *fast* mode ㉔. Subsequently, the response $\mathbf{r}$ is sent back to the Server $\mathcal{S}$, ㉕ the *WisecrMode* flag is cleared and ⑲ the device is reset.

Notably, a power-loss event during the control flow will result in a reset and rebooting of the token T and a transition out from the firmware update state. Most significantly, in such an occurrence, the immutable bootloader functionality is preserved. Although the loss of state implies that a Server $\mathcal{S}$ must reattempt the secure update, we prevent the token from being left in a vulnerable state to inherently mitigate security threats posed in the power *off* state [22].

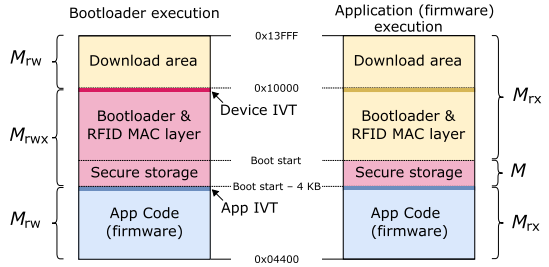

Fig. 10. Memory Protection Unit (MPU) segmentation diagram and memory arrangement during bootloader execution (left) and application code execution (right). Due to hardware limitations, only 3 segments can be defined, and the secure storage area must be at least 1 KiB in size due to segment boundary alignment requirements [45]

4 End-to-End Implementation and Experiments

This section elaborates on software tools, the end-to-end implementation of *Wisecr* and extensive experiments conducted to systematically evaluate the performance of *Wisecr*.

4.1 End-to-End Implementation

We implement a bootloader and a *Server Toolkit* to wirelessly and simultaneously update multiple CRFID transponders using *commodity* networked RFID hardware and *standard* protocols. We realize the *Wisecr* scheme in Fig. 4 and the identified security property requirements. Specifically, we employ MSP430FR5969 MCU based WIPS5.1LRG CRFID devices. The MSP430FR5969 microcontroller has a 2 KiB SRAM and a 64 KiB FRAM internal memory. *The implementation is a significant undertaking* and includes software development for the CRFID as well as the RFID reader and backend services for the server update mechanisms. Given that we have open sourced the project, we describe: i) the *bootloader* in Section 4.2; and ii) the Sever Toolkit and the protocols for Host-to-RFID-reader and Reader-to-CRFID-device communications for implementing *Wisecr* over *EPC Gen2* in Appendix B, available in the online supplemental material.

4.2 Bootloader and Memory Management

Wisecr Bootloader. The immutable bootloader supervises firmware execution while needing to be compatible with standard protocols, we developed our bootloader based on the Texas Instrument's recent framework for wireless firmware updates—*MSP430FRBoot* [44] —and the code base from recent work [17] employing the framework. Such construction ensures a standard tool chain for compilation and update of new firmware whilst the usage of industry standards is likely to increase adoption in the future. We compile the firmware code into ELF (Executable and Linkable Format) made up of binary machine code and linker map specifying the target memory allocation—the memory arrangement is illustrated Fig. 10.

Wisecr Memory Management. For implementing the Secure Storage component, several different mechanisms are available (as summarized in Appendix A, available in the online supplemental material). In *Wisecr*, we select to use the *Runtime access protection (e.g. using MPU segments at run-time)* In this scheme, secure storage is available on device boot-up, but is locked (until next boot-up) by the bootloader before

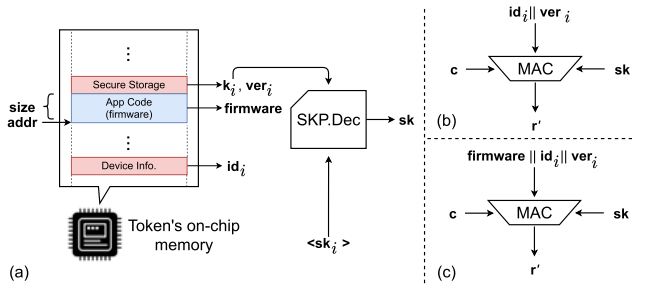

Fig. 11. (a) Token memory and allocations, (b) *fast* mode, (c) *elaborate* mode.

any application code is executed. We employed this method and the implemented memory arrangement is described in Fig. 10. Switching to privilege mode is done by the bootloader on boot up, and the applications do not need to make any specific modifications. If an application attempts to access a privileged resource, e.g. overwriting the bootloader segment, a power-up clear (PUC) will be triggered by the MPU to cause reboot, before any risky operation.

Importantly, any power-on reset, regardless of the cause, will result in a reboot and the execution of the bootloader before transferring control to application code—see Application execution stage in Fig. 9. Prior to entering the application execution stage, the bootloader enforces the MPU policy for application execution (see ③). This step is necessary with the target MCU used in our implementation as it only allows defining three memory segments for protection. Thus, we are able to rely on the very limited protections provided by the MCU to achieve the security objectives of a trusted bootloader; recall, in our threat model, the bootloader is trusted, and we realize this in practice by ensuring that the bootloader provisioned is immutable where the secrets stored on device must remain inaccessible and immutable to the application firmware. Now, it is infeasible to revoke the policy enforced during the application execution stage, unless the device is power-cycled. But, the device will enter the bootloader stage after a reboot and MPU protections will be re-enabled prior to executing any application code. We summarize the memory protections developed using the limited MPU configurations to realize *Wisecr* security requirements in bootloader and application execution modes in Section 3.2; therein, we also discuss alternative methods to realize such protections to ensure the generalization of *Wisecr* to other MCUs.

Remote Attestation. As shown in Fig. 11a Remote Attestation routine is an integral part of the immutable bootloader, it has access to all memory segments: Token device key $\mathbf{k}_i$, version number $\mathbf{ver}_i$, tag serial number $\mathbf{id}_i$ and the installed user application code *firmware*. A challenge $\mathbf{c}$ is a one-time use random string that is generated by the Server $\mathcal{S}$. The $\mathbf{r}'$ is the report to be transferred back to the Server. On receiving an encrypted session key $\langle \mathbf{sk}_i \rangle$, the Token decrypts it with SKP.Dec and obtains the session key $\mathbf{sk}$. In the *fast* mode illustrated in Fig. 11b, the response $\mathbf{r}'$ is calculated based on $\mathbf{id}_i$ and $\mathbf{ver}_i$. In contrast, the response $\mathbf{r}'$ is computed over an entire memory segment, such as the *firmware*, in the *elaborate* mode as depicted in Fig. 11c.

4.3 *Wisecr* Implementation Overheads

We first evaluate the execution and implementation overhead of each *Wisecr security functional block*; results[4] are summarized in Table 3. The execution overhead is measured in terms of clock cycles for installing a firmware of 240 Bytes. The implementation overhead is measured in memory usage, consisting of ROM for code and constants and RAM for run-time state. Secondly, we assess the necessary hardware modules such as hardware AES accelerator (HWAES), analog-to-digital converter (ADC) and Timer. All functional blocks are implemented in platform independent C source files. We summarize the implementation costs for each stage; the most significant implementation and execution overhead is from the Validation stage because of the computationally heavy MAC() function—recall Remote Attestation is an optional stage.

In Table 3, we also compare with SecuCode[5]; presently, the only CRFID firmware update scheme considering security—notably, the scheme only prevents malicious code injection attacks and is designed to update a single device at a time as highlighted in the method comparison Table 4. In summary, *Wisecr* consumes 49.97% more clock cycles, 40.95% more ROM and 97.43% more RAM space than the SecuCode. However, *Wisecr* can update multiple devices simultaneously to attain a performance advantage as we demonstrate in Section 4.3 because SecuCode only updates one device at a time. In addition, Wisecr provides IP protection (encryption of the firmware transmitted over the insecure wireless channel) and validates firmware installation. As expected, the performance improvements and additional security features are achieved at an increased implementation overhead.

4.4 Experiment Designs and Evaluations

We employed four WIPS5.1LRG CRFID devices and an Impinj R420 RFID reader with a 9 dBic gain antenna to communicate with and energize the CRFID devices. All of the experiments are conducted with the RFID reader feeding the antenna with 30 dBm output power. Given that, the recent *EPC Gen2* V2.0 [35] security commands such as the `SecureComm` are not yet widely supported by commercial RFID readers, like the Impinj Speedway R420, we map the unsupported *EPC Gen2* command to the `BlockWrite` command as in [17]. We summarize our evaluations below:[6]

- We evaluate performance—latency and throughput in Section 4.5—and compare with: i) SecuCode [17], the scheme only prevents malicious code injection attacks and is designed to update a single device at a time as shown in the method comparison Table 4; ii) *Wisecr* intentionally set to sequential mode—broadcasting/updating one token at a time—termed *Wisecr* (Seq), to assess the gains from *Wisecr* broadcasting firmware to

4. Tested under the CCS 9.0.1.0004 development environment, with compiler TI v18.12.1.LTS Optimization settings: -O = 3; -opt_for_speed = 5.

5. In [17], the device key is derived from SRAM PUF responses. In our comparison, we opt for a device key in the protected NVM instead to make a fair overhead comparison. Further, the key derivation module only adds a fixed overhead at start-up.

6. Notably, our extensive experiments to evaluate techniques and modules we developed are detailed in the Appendices with references in the main article

TABLE 3
Wisecr Implementation and Execution Overheads

	Clock Cycles	Memory Usage (Bytes)		HW Requirement
		ROM	RAM	
Functional blocks				
$\mathbf{sk} \leftarrow \mathsf{SKP.Dec}_{\mathbf{k}i}(\langle \mathbf{sk}_i \rangle)$ (For a 16-Byte block)	772	706[2]	62	HW-AES
$\mathbf{firmware}^{(j)} \leftarrow \mathsf{SKP.Dec}_{\mathbf{sk}}(\langle \mathbf{firmware}^{(j)} \rangle)$ (For a 16-Byte block)	772	690	62	HW-AES
$\mathbf{s}_i \leftarrow \mathsf{MAC}_{\mathbf{sk}}(\mathbf{firmware}\|\|\mathbf{ver}_i\|\|\mathbf{nver})$	26,698[1]	2,374	72	HW-AES
$\mathbf{Vt}_i \leftarrow \mathsf{SNIFF}(t)$	1,530	276	12	ADC
$\mathsf{PAM.Enter}(\mathrm{t}_{LPMi}, \mathrm{t}_{activei})$ and $\mathsf{PAM.Exit}()$[3]	51	86	6	Timer
Setup MPU for bootloader execution	76	58	0	MPU
Setup MPU for application execution	91	60	0	MPU
Secure Storage (used for $\mathbf{k}_i$, $\mathbf{id}_i$ and $\mathbf{ver}_i$)	0	19	0	MPU
Stages				
Security Association	2,302	982	74	ADC, HW-AES
Secure Broadcast	11,604[1]	760	68	HW-AES, Timer
Validation	26,724[1]	2,390	77	HW-AES, Timer
Remote Attestation (*elaborate* mode)	18,360[4]	3,172	80	HW-AES
Remote Attestation (*fast* mode)	5,574	3,172	80	HW-AES
Total				
Wisecr (excluding Remote Attestation, for the **pilot** Token)	40,630[1]	3442	154	All of the above
SecuCode (using fixed key, updates a single token)	27,092[1]	2442	78	HW-AES, Timer

[1] *For a typical 240-Byte firmware.*
[2] *16 Bytes of this value incorporates the device key* $\mathbf{k}_i$ *in secure storage.*
[3] *A single PAM execution, as defined in Fig. 6.*
[4] *4,397 clock cycles for the setup routine–a constant overhead for any block size, 1,060 clock cycles for each 16-Byte block and* $n + 2$ *clock cycles for an* n*-Byte padding to form a 16-Byte block, if required.*

multiple devices simultaneously; and iii) non-secure multi-CRFID dissemination scheme, Stork [7] (see Section 5.1)
- We evaluate the efficacy of our Power Aware Execution Model (PAM) and Pilot Token selection method (results in the Section. 3.2.2).
- We demonstrate *Wisecr* in an end-to-end implementation (see Section 4.6 and demonstration video at https://youtu.be/GgDHPJi3A5U)

4.5 Performance Evaluation and Results

We use two performance metrics: i) end-to-end *latency*; and ii) *throughput* (bits per second). Latency is the time elapsed from the Server Toolkit transmitting the firmware to an RFID reader using LLRP commands to the time the Server Toolkit confirms the acknowledgment of successfully updating *all chosen tokens*. Throughput is the total data transferred over the latency, where $Throughput\ (bps) = (|\mathbf{firmware}| \times Num.\ of\ Tokens).Latency$. Our evaluation is carried out under three different controlled variables: i) distance; ii) number of tokens; and iii) firmware size. In each experiment, only one variable is changed and our results are collected over 100 repeated measurements, with outliers outside of 1.5 times upper and lower quartiles removed.

First, we transfer the same firmware code (a 407 Byte accelerometer service) to four target devices located at distances from 20 cm to 40 cm, covering good to poor powering channel states, to evaluate the *stability* of the *Wisecr* scheme. As expected, we can observe in Fig. 12a that the performance of all three schemes downgrades with increasing distance. Notably, *Wisecr* and SecuCode outperforms the *Wisecr* (Seq). Because, *Wisecr* (Seq) incurs a larger overhead through the need for executing a Security Association stage to setup the broadcast security parameters, for each device. Further, SecuCode (without IP protection) does not need to execute the power intensive firmware decryption operations, and thus is less likely to encounter power-loss and update session failure or require the extra time necessary by *Wisecr* to decrpyt the firmware. As expected, *Wisecr* outperforms albeit the added security functions provided in comparison to SecuCode.

TABLE 4
Comparison With Related Studies

Work	Multiple devices	Power-loss mitigation		Security			Public source code release
		Commun. Energy Reduction	Adaptive IEM	Prevent Malicious Code Injection	Prevent IP Theft	Code instal. attest.	
Wisent [18]	✘	✘	✘	✘	✘	✘	✓
R^3 [21]	✘	✘	✘	✘	✘	✘	✘
Stork [7]	✓	✓	✘	✘	✘	✘	✓
SecuCode [17]	✘	✘	✘	✓	✘	✘	✓
Wisecr (Ours)	✓	✓	✓	✓	✓	✓	✓

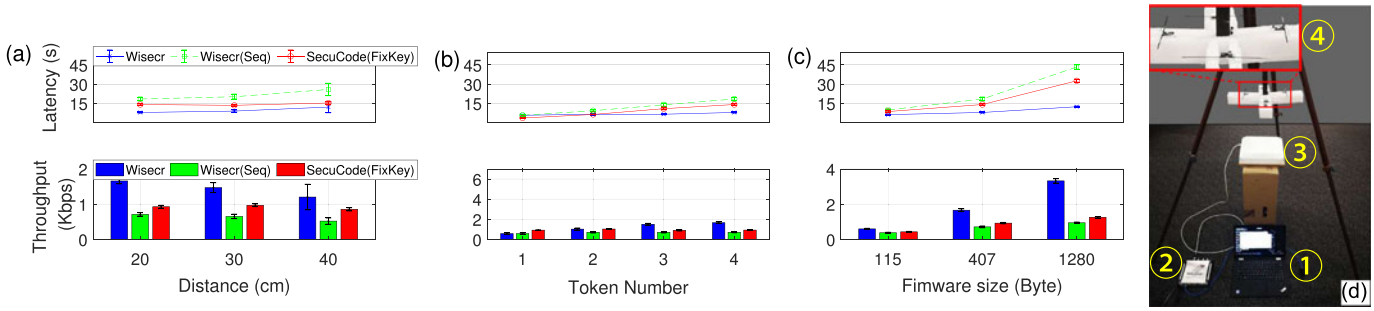

Fig. 12. Comparing *Wisecr*, *Wisecr* (Seq)—*Wisecr* operating in sequential mode, SecuCode [17] *with no secure broadcast* support under increasing: (a) Distance (4 Tokens with a 407 Byte firmware), (b) Number of Tokens (at 20 cm with 407 Byte firmware) and (c) Firmware size (at 20 cm with 4 Tokens). The experimental setup illustrated in (d) shows ① a host PC, ② a RFID reader, ③ a reader antenna and ④ four CRFID devices mounted on a foam frame, supported by a wooden tripod.

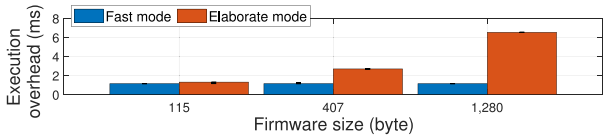

Fig. 13. The execution overhead of the attestation function with respect to different firmware sizes in an end-to-end experiment setting. Mean values over 100 repeated measurements are plotted.

Second, following the method and metrics in Stork [7], we tested the performance of the schemes by increasing the number of CRFID devices to be updated. We can see in Fig. 12b that the latency of both *Wisecr* (Seq) and SecuCode increases linearly while *Wisecr* remains relatively steady regardless of the number of devices to update. Consequently, *Wisecr* exhibits significantly improved throughput as the number of device increases. Further, we examine performance under increasing firmware size: 115 Bytes; 407 Bytes (a sensor service code); and 1280 Bytes (a computation code firmware). Efficacy of broadcasting to multiple devices simultaneously is validated by latency and throughput performance detailed in Fig. 12c.

Third, all three candidates show improved efficiency as larger firmware sizes are transmitted; *Wisecr*'s significantly better efficiency can be attributed to the broadcast method.

We also conducted an evaluation of the execution overhead (latency) introduced by the remote attestation routines. Since remote attestation is performed on one singulated token at an instance, we only examine its impact on one individual device. The latency can be aggregated for multiple devices. As illustrated in Fig. 13, the remote attestation in *fast* mode always completes in 1.1 ms, regardless of the firmware size. While the time required to complete the *elaborate* mode scales from 1.3 ms to 6.5 ms with respect to firmware sizes from 115 Bytes to 1280 Bytes.

4.6 Case Study

As shown in Fig. 14, we employ four CRFID devices embedded in a 3D printed unmanned aerial vehicle (UAV) rotor prototype. *The CRFID devices are factory programmed with firmware for monitoring the centrifugal load and the wired programming interface is disabled prior to embedding and deployment. All four CRFID devices are required to be updated wirelessly and securely with code for monitor the blade flapping vibration in a wind tunnel test.*

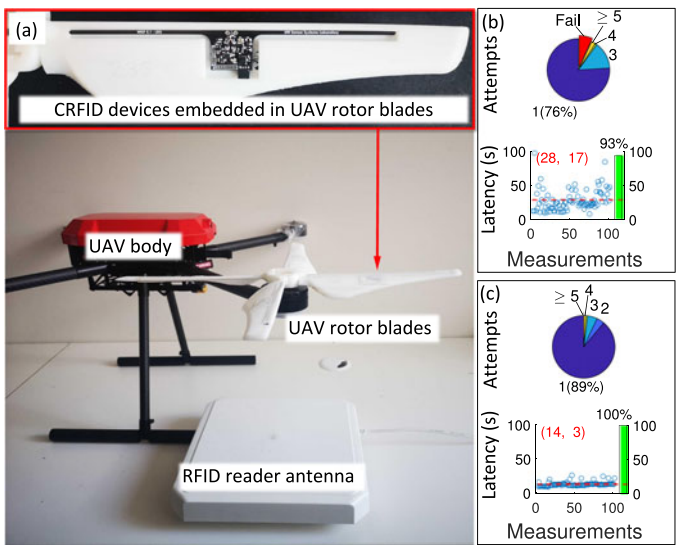

Fig. 14. (a) Application scenario: wirelessly updating the firmware of four CRFID sensor devices embedded in an unmanned aerial vehicle (UAV) rotor prototype[7]. With *Wisecr*, we simultaneously, securely and wirelessly updated the four devices with a 391-Byte code. Experimental results when: (b) the rotor assembly is attached to the UAV; and (c) when the rotor assembly is free standing over the antenna. The green bar graph denotes the percentage of successful updates to all 4 devices.

We employed *Wisecr* to simultaneously, securely and wirelessly update the four CRFID sensor devices with the 391-Byte firmware code. The experimental results summarized in Fig. 14 show successful updates for 100 repeated attempts in two settings: i) when the rotor is attached to the UAV *Wisecr* updated all of the devices in the first attempt in 76% of the time with an average latency of 28.12 seconds; and ii) when the rotor assembly is free standing, *Wisecr* updated all of the devices in the first attempt in 89% of the time with an average latency of 14.27 seconds[8].

7. The UAV rotor aerodynamic model is adopted and modified from: https://grabcad.com/library/4-bladed-propeller-experimental-1

8. Although previous studies have performed experiments with static tags, we also attempted to update the tokens while the rotor blades are mobile by rotating them at approximately 1 RPM. We were able to update tokens over 50% of the time. As expected, the changing powering conditions dramatically reduced the update rate (see details in Appendix I, available in the online supplemental material)

A demonstration of the secure firmware update process in using our end-to-end implementation illustrated in Fig. 14 is available at https://youtu.be/GgDHPJi3A5U. We open source the complete source code for *Wisecr* and related tools at https://github.com/AdelaideAuto-IDLab/Wisecr.

4.7 Security Analysis

Under the threat model in Section 2.1, we analyze *Wisecr* security against: i) IP theft (code reverse engineering); ii) malicious code injection attacks under code alteration, loading unauthorized code, loading code onto an unauthorized device, and code downgrading; and iii) incomplete code installation.

IP Theft or Code Reverse Engineering. The wirelessly broadcasted firmware is encrypted using a one-time 128-bit broadcast session key $\mathbf{sk}_i$. Without the knowledge of the broadcast session key, the adversary $\mathcal{A}$ is unable to obtain the plaintext **firmware** through passive eavesdropping. Once the firmware is decrypted on the token, the binaries cannot be accessed outside the immutable bootloader. Therefore, the probability of a successful IP theft is now determined by the security provided by the size of the selected keys—we employ 128-bit keys in the *Wisecr* implementation.

Loading Unauthorized Code. The security property of a MAC tag $\mathbf{s}_i \leftarrow \mathsf{MAC}_{\mathbf{k}i}(\mathbf{firmware}||\mathbf{ver}_i||\mathbf{nver})$ ensures that the adversary $\mathcal{A}$ cannot commit a malicious firmware injection attack without the knowledge of the session key **sk**—recall that the sharing of **sk** is protected through a secure channel established with a token specific private key $\mathbf{k}_i$. Hence, only the Server with the session key **sk** is able to produce a valid MAC tag to a token T. Hence, the probability of launching a successful attack by the adversary $\mathcal{A}$ is limited to the probability of successfully obtaining the device key $\mathbf{k}_i$ and/or the session key **sk** in a man-in-the-middle attack to successfully construct a MAC tag and disseminate a malicious firmware that will be correctly validated by the target tokens. Therefore, without knowledge of $\mathbf{k}_i$ or **sk**, the probability of fooling a token T to inject a malicious firmware by $\mathcal{A}$ is no more than the brute-force attack probability 2^{-128} determined by the key size $|\mathbf{k}_i| = 128$.

Code Alteration. For each firmware update, a MAC tag $\mathbf{s}_i \leftarrow \mathsf{MAC}_{\mathbf{k}i}(\mathbf{firmware}||\mathbf{ver}_i||\mathbf{nver})$ is used to verify code integrity. Therefore, without knowledge of the session key **sk**, the adversary $\mathcal{A}$ can not generate a valid MAC tag for an altered firmware. Any attempted changes to the firmware during the code dissemination will be detected and, thus, firmware discarded by the token.

Loading Code Onto an Unauthorized Device. Each authorized device i maintains a unique and secret device specific key $\mathbf{k}_i$ only known to the Server $\mathcal{S}$. Therefore, an unauthorized device cannot decrypt the session key $\mathbf{sk}_i$ to install a recorded firmware encrypted with $\mathbf{sk}_i$ of an authorized device.

Code Downgrading. The adversary $\mathcal{A}$ can attempt to downgrade the firmware to an older version to facilitate exploitation of potential vulnerabilities in a previous distribution. This can be attempted through a replay attack by impersonating the Server. However, for each firmware update, a specific MAC tag: $\mathbf{s}_i \leftarrow \mathsf{MAC}_{\mathbf{k}i}(\mathbf{firmware}||\mathbf{ver}_i||\mathbf{nver})$ is generated based on two monotonically increasing version numbers: $\mathbf{ver}_i$ and **nver**. Without direct access to modify the Secure storage contents in region M to re-write the current version number of a device to an older version, the attacker cannot replay a recorded MAC tag from a previous session to force a token to accept an older firmware. By virtue of the monotonically increasing version numbers, and the secure MAC primitive protected by the session key **sk**, a successful software downgrade using a replay attack cannot be mounted by the adversary $\mathcal{A}$.

Incomplete Firmware Installation. In a man-in-the-middle attack, the adversary $\mathcal{A}$ may attempt to prevent a new firmware installation and spoof a device response with an updated version number during an interrogation—the version number is public information and sent in plaintext as shown in Fig. 4. Hence, a token may be left executing an older firmware version with potential firmware vulnerabilities. The Server can verify that a specific disseminated firmware deployed by the token is the same as the one issued by the Server by performing a remote attestation. In our *Wisecr* scheme, we have considered an optional remote attestation stage (detailed in Section 2.2) to allow the Server to verify the firmware installation on a given token. In general, both code installation and attestation are bounded to the device key $\mathbf{k}_i$, thus, it is infeasible to fool the Server without knowledge of the device key.

Related-Key Differential Cryptanalysis Attack. In our security association stage, the session key **sk** is encrypted multiple times with different device keys $\mathbf{k}_i$. A related-key differential cryptanalysis of round-reduced AES-128 is demonstrated in [46]. The prerequisite to mount such an attack is that the attacker knows or is able to choose a relation between several secret keys and gain access to both plaintext and ciphertext [47]. In our approach, as mentioned in Section 2.1, $\mathbf{k}_i$ chosen by the server are *i.i.d.*, and only the ciphertext is publicly available where the attacker is not able to force the server to encrypt a plaintext of their choosing. Therefore related-key attacks cannot be mounted under our threat model. Even if such an attack is possible, to the best of our knowledge, there is no successful full-round related-key differential cryptanalysis attack against AES-128, which serve as the SKP function in our implementation.

5 Related Work and Discussion

Here, we discuss related works in the area of *wireless code dissemination to CRFID devices*. Notably, the topic of wireless code updating has been intensively studied for battery powered Wireless Sensor Network (WSN) nodes. For example, a recent study by Florian *et al*. [29] proposed updating the firmware and peer-to-peer attestation of multiple mesh networked IoT devices. Physical unclonable functions [17] and blockchains [48] have also been utilized in code dissemination But, such schemes [29], [48] are designed for *battery powered* WSN nodes featuring *device-to-device communication capability*, and *supervisory control of operating systems* (e.g TinyOS), *absent* in batteryless CRFID devices (see challenges in Section 1). Further, batteryless devices operate under extreme energy and computational capability limitations. In particular, harvested energy is intermittent and limited and thus devices face difficulties supporting security functions

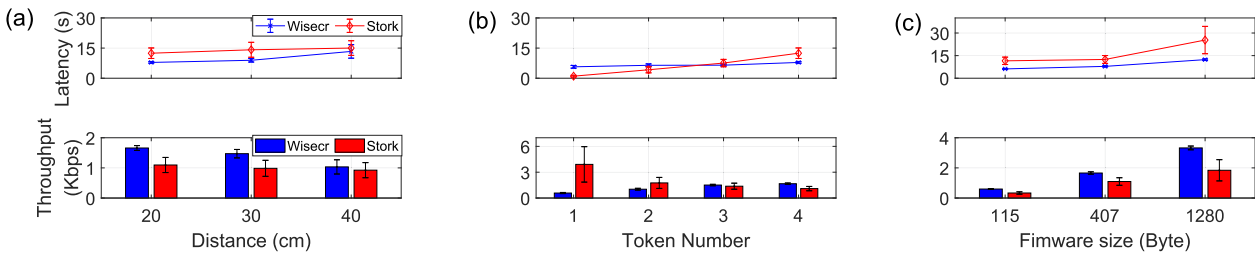

Fig. 15. We repeated the firmware update process for *Wisecr* (ours) and Stork to compare between Stork (providing *no security properties*) and our *Wisecr* scheme under three different test settings: (a) increasing operational range; (b) increasing number of tokens, and (c) payload sizes.

such as key exchange using Elliptic Curve Diffie-Hellman used in [29]. Therefore, we focus on research into wireless dissemination schemes for CRFID like devices in our discussions. Further, we also discuss and compare our work with existing wireless methods developed for CRFID to highlight the aim of our scheme to fulfill unmet security objectives for simultaneous wireless firmware update to multiple CRFIDs.

Wireless Code Dissemination to CRFID. Given the recent emergence of the technology, there exists only a few studies on code dissemination to CRFID devices. *Wisent* from Tan *et al.* [18], and R^3 from Wu *et al.* [21] demonstrate a wireless firmware update method for CRFIDs. Subsequently, Aantjes *et al.* [7] proposed *Stork*, a fast multi-CRFID wireless firmware transfer protocol. Stork forces devices to ignore the RN16 handle in down-link packets—a form of promiscuous listening—to realize a logical broadcast channel in the absence of *EPC Gen2* support for a broadcast capability; RN16 specifies the designated packet receiver. This technique enables Stork to simultaneously program multiple CRFIDs to achieve *fast* firmware dissemination. Our Pilot-Observer mode (Section 3.2.2) is based on a similar concept, but instead of always selecting the first seen device, as in Stork, we strategically elect the token with the lowest V_t as the *Pilot* to achieve higher broadcast success. Brown and Pier from Texas Instrument (TI) developed *MSPBoot* [49] in late 2016. They demonstrate an update using a UART or SPI bus to interconnect a MSP430 16-bit RISC microcontroller and a CC1101 sub-1GHz RF transceiver.

However, none of the schemes, *Wisent*, R^3, Stork and the work in [49], considers security when wirelessly updating firmware. SecuCode [17] took the first step to prevent malicious code injection attacks where a single CRFID device is updated. But SecuCode lacks scalability and performs device by device updates, cannot protect the firmware IP, and provides no validation of firmware installation.

Remote Attestation. Attestation enables the Server to establish trust with the token's hardware and software configuration. Existing mainstream remote attestation methods are unsuitable for a practicable realization in the context of intermittently powered, energy harvesting platforms we focus on. For example, peer-to-peer attestation adopted in [29] is impractical in the CRFID context due to the lack of a device-to-device communication channel. In boot attestation [50], a public-key based scheme is proposed to fit ownership and third party attestation, however public-key schemes are too computationally intensive for the batteryless devices we consider. Recently, a publish/subscribe mechanism based asynchronous attestation for large scale WSN is presented in SARA [51], however, such a publish/subscribe paradigm is yet to be supported over a standard *EPC Gen2* protocol. In contrast, our lightweight-remote-attestation mechanisms (see Section 2.2) are inspired from the recent study in [19] where methods are designed for resource limited platforms and require no additional building blocks besides those necessary for *Wisecr*.

Comparisons. Table 4 summarizes a comparison of *Wisecr* with wireless code dissemination studies *specific to passive CRFID devices*. *Wisecr* is the only *secure* scheme (prevent IP theft and malicious code injection, and provide attestation of firmware installation) for *simultaneous* update of passively powered devices. We have also extensively compared the performance of *Wisecr* with the *non-secure* code update method of *Stork* to provide an appreciation for the security overhead in an end-to-end implementation below.

5.1 Comparison With Stork (Insecure Method)

We extensively compared the performance of our secure *Wisecr* scheme with Stork [7]. Both methods aim to offer multiple CRFID wireless firmware updates, but security objectives are not considered by Stork. Comparisons are carried out under three different test settings:

- Four different operating ranges from 20 to 50 cm.
- Updating 1 to 4 tags concurrently in-field to evaluate reduced latency to update (or improvements to scalability) (as in Stork, tags are at 20 cm from the antenna).
- Consider three different firmware sizes (as in Stork, tags are at 20 cm from the antenna).

We measured two performance metrics: i) latency; and ii) throughput as defined in Section 4.5 for each test setting. For both Stork and *Wisecr*, we used the same binary files generated from the same compilation. The settings in the *Wisecr* Server Toolkit are: 10 attempts per broadcast run; lowest voltage $\mathbf{Vt_i}$ pilot token selection method; Send Mode is set to Broadcast. Meanwhile, the settings in Stork are: BWPayload is set to throttle; Update Only is set to True; Reprogramming Mode is set to Broadcast; Compression is Disabled (as compression simply reduces the size of the firmware, we did not enable this option). Results are mean values from 100 repeated protocol update runs. Comparison results from our experiments are detailed in Fig. 15:

- From Fig. 15a: when powering channel state is reasonable i.e. from 20 cm to 40 cm, *Wisecr* outperforms Stork, as *Wisecr* does not need per-block checking and relies on the pilot token selection method to improve broadcast performance. However, Stork performs better than *Wisecr* at 50 cm when the

powering channel condition is very poor because Stork is able to continue to transmit the firmware across power failures since the stork scheme does not need to meet any security requirements and therefore is able to send firmware payload in plaintext from the last correctly received payload. In contrast *Wisecr* gracefully fails when a power interruption cannot be prevented and all states (such as the session key) are lost and a new broadcast attempt must be made by the Server Toolkit.

- From Fig. 15b: *Wisecr* exhibits significantly better latency and throughput as the number of tokens increases.
- From Fig. 15c: both protocols show improved efficiency as larger payload/firmware sizes are transmitted. Notably, *Wisecr* exhibits much better efficiency attributed to the significantly reduced latency experienced during the broadcast method.

The gains in performance and faster updates to many devices achieved by *Wisecr* can be attributed to: i) our pilot token selection method to drive the protocol where a significantly large proportion of in-field tokens are updated in a single attempt; and ii) our validation method of complexity $O(n \cdot 1)$ where n is the number of tokens—although computationally intensive—is more lightweight than the read back mechanism of Stork, $O(n \cdot k)$ where k is the code size, relying on the narrow band communication channel employed by CRFID devices.

6 Conclusion and Future Work

Our study proposed and implemented the first secure and simultaneous wireless firmware update to many RF-powered devices with remote attestation of code installation. We explored highly limited resources and innovated to resolve security engineering challenges to implement *Wisecr*. The scheme prevents malicious code injection, IP theft, and incomplete code installation threats whilst being compliant with standard hardware and protocols. *Wisecr* performance and comparisons with state-of-the-art through an extensive experiment regime have validated the efficacy and practicality of the design, whilst the end-to-end implementation source code is released to facilitate further improvements by practitioners and the academic community.

Limitations and Future Work: Our study is not without limitations. Although the *Wisecr* reasonably assumes (as in R^3 [21] and *Stork* [7]) that the devices are at relatively constant distances from an RFID reader antenna during the short duration firmware update process (approximately 16 s in the application demonstration), occasionally this assumption may not hold. For example, when re-programming devices on a conveyor belt, it is desirable to not delay the movement of the tags and to re-program whilst distances are changing. Development of a solution to the problem in the context of resource limited devices operating over a highly constrained air interface protocol is a challenging problem and an interesting direction for future work.

Additionally, every `BlockWrite` command used to broadcast the firmware contains a CRC-16 field for error detection [35]. In our Pilot-Observer mode, only the pilot-detected CRC errors are indicated to the Host; future work could consider the problem of broadcast reliability further. Notably, developing a method is non-trivial given the limitations of the *EPC Gen2* air interface protocol.

In our secure update method, the *pilot* token and *observer* tokens do not explicitly synchronize progress due to the constrained air interface protocol not supporting token-to-token communications as well as the extremely limited available power. This also makes it difficult for the server to immediately, i.e. during the *Secure Broadcast* stage, to detect an observer token that has de-synchronized. Any synchronization would need to involve the Host performing this function, such as reading back of memory contents from a token's download region and re-transmitting missing packets, at set intervals [7]. Hence, developing an explicit synchronization method in the context of a secure broadcast method forms an interesting direction for future work.

Further, it will be interesting to consider alternative mechanisms for validating the installation of firmware, such as the exploitation of the reply signal to the final packet delivering firmware to tokens. We leave this interesting direction to explore for future work.

References

[1] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," *Proc. Summit Adv. Program. Lang.*, vol. 71, pp. 8:1–8:14, 2017.

[2] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-based battery-free programmable sensing platform," *IEEE Trans. Instrum. Meas.*, vol. 57, no. 11, pp. 2608–2615, Nov. 2008.

[3] H. Zhang, J. Gummeson, B. Ransford, and K. Fu, "Moo: A batteryless computational RFID and sensing platform," Comput. Sci. Technol., Univ. Massachusetts, Boston, Mass., Tech. Rep. UM-CS-2011–020, 2011.

[4] Farsens, "Pyros-0373 UHF RFID battery-free temperature sensor tag," Accessed: Mar. 19, 2022. [Online]. Available: http://www.farsens.com/en/products/pyros-0373/

[5] A. J. Bandodkar *et al.*, "Battery-free, skin-interfaced microfluidic/electronic systems for simultaneous electrochemical, colorimetric, and volumetric analysis of sweat," *Sci. Adv.*, vol. 5, no. 1, 2019, Art. no. eaav3294.

[6] R. L. Shinmoto Torres, R. Visvanathan, D. Abbott, K. D. Hill, and D. C. Ranasinghe, "A battery-less and wireless wearable sensor system for identifying bed and chair exits in a pilot trial in hospitalized older people," *PLOS One*, vol. 12, no. 10, pp. 1–25, 2017.

[7] H. Aantjes, A. Y. Majid, P. Pawełczak, J. Tan, A. Parks, and J. R. Smith, "Fast downstream to many (computational) RFIDs," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2017, pp. 1–9.

[8] D. Marasová, P. Koščák, N. Staricna, S. Mako, and D. Matisková, "Digitization of air transport using smart tires," in *Proc. New Trends Aviation Develop.*, 2020, pp. 164–167.

[9] S. Yang, M. Crisp, R. V. Penty, and I. H. White, "RFID enabled health monitoring system for aircraft landing gear," *J. Radio Freq. Identification*, vol. 2, no. 3, pp. 159–169, 2018.

[10] M. D. Santonino, III, C. M. Koursaris, and M. J. Williams, "Modernizing the supply chain of airbus by integrating RFID and blockchain processes," *Int. J. Aviation Aeronaut. Aerosp.*, vol. 5, no. 4, 2018, Art. no. 4.

[11] W. Li, H. Song, and F. Zeng, "Policy-based secure and trustworthy sensing for internet of things in smart cities," *Internet of Things J.*, vol. 5, no. 2, pp. 716–723, 2017.

[12] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Net. App.*, vol. 24, no. 3, pp. 796–809, 2019.

[13] A. Parks, "WISP 5," 2016, Accessed: Mar. 19, 2022. [Online]. Available: https://github.com/wisp/wisp5

[14] Z. Ning and F. Zhang, "Understanding the security of arm debugging features," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 602–619.

[15] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "BadBluetooth: Breaking android security mechanisms via malicious bluetooth peripherals," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.
[16] FAA, "Airworthiness approval of installed radio frequency identification (RFID) tags and sensors," Nov. 2018, Accessed: Mar. 19, 2022. [Online]. Available: www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentid/1034630
[17] Y. Su, Y. Gao, M. Chesser, O. Kavehei, A. Sample, and D. Ranasinghe, "SecuCode: Intrinsic PUF entangled secure wireless code dissemination for computational RFID devices," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 4, pp. 1699–1717, Jul./Aug. 2021.
[18] J. Tan, P. Pawełczak, A. Parks, and J. R. Smith, "Wisent: Robust downstream communication and storage for computational RFIDs," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[19] D. Dinu, A. S. Khrishnan, and P. Schaumont, "SIA: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers," in *Proc. IEEE Int. Symp. Hardware Oriented Secur. Trust*, 2019, pp. 208–217.
[20] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, vol. 1, 2017, Art. no. 96.
[21] D. Wu, L. Lu, M. J. Hussain, S. Li, M. Li, and F. Zhang, "R3: Reliable over-the-air reprogramming on computational RFIDs," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, 2018, Art. no. 9.
[22] A. S. Krishnan and P. Schaumont, "Exploiting security vulnerabilities in intermittent computing," in *Proc. Int. Conf. Secur., Privacy Appl. Cryptogr. Eng.*, 2018, pp. 104–124.
[23] Y. Su and D. C. Ranasinghe, "Leaving your things unattended is no joke! memory bus snooping and open debug interface exploits," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, 2022, pp. 643–648.
[24] Z. Liu, H. Seo, Z. Hu, X. Hunag, and J. Großschädl, "Efficient implementation of ECDH key exchange for MSP430-based wireless sensor networks," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2015, pp. 145–153.
[25] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede, "Efficient ring-LWE encryption on 8-bit AVR processors," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 663–682.
[26] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: Arming trustzone with user-space enclaves," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.
[27] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 83–97.
[28] C. Jin and M. van Dijk, "Secure and efficient initialization and authentication protocols for SHIELD," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 1, pp. 156–173, Jan./Feb. 2019.
[29] F. Kohnhäuser and S. Katzenbeisser, "Secure code updates for mesh networked commodity low-end embedded devices," in *Proc. Springer Eur. Symp. Res. Comput. Secur.*, 2016, pp. 320–338.
[30] W. Feng, Y. Qin, S. Zhao, Z. Liu, X. Chu, and D. Feng, "Secure code updates for smart embedded devices based on PUFs," in *Proc. Int. Conf. Cryptogr. Net. Secur.*, 2017, pp. 325–346.
[31] F. Piessens and I. Verbauwhede, "Software security: Vulnerabilities and countermeasures for two attacker models," in *Proc. Conf. Des. Automat. Test Eur.*, 2016, pp. 990–999.
[32] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "RefinedC: Automating the foundational verification of C code with refined ownership types," in *Proc. Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 158–174.
[33] A. Levy *et al.*, "Multiprogramming a 64kb computer safely and efficiently," in *Proc. Symp. Operating Syst. Princ.*, 2017, pp. 234–251.
[34] A. Aysu, E. Gulcan, D. Moriyama, P. Schaumont, and M. Yung, "End-to-end design of a PUF-based privacy preserving authentication protocol," in *Proc. Springer Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 556–576.
[35] G. EPCglobal, "Inc.,"EPCTM radio-frequency identity protocols class-1 generation-2 UHF RFID protocol for communications at 860MHz-960MHz Version 2.0.1," EPCGlobal Inc., Lawrenceville, NJ, 2015, Accessed: Mar. 2, 2022. [Online]. Available: https://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard%.pdf
[36] H. T. Friis, "The free space transmission equation," *Proc. IRE*, vol. 34, 1946, Art. no. 254.
[37] P. Thanigai and W. Goh, "MSP430 FRAM quality and reliability," Texas Instrum. Inc., Dallas, TX, USA, Tech. Rep., 2014. Accessed: Oct. 4, 2022. [Online]. Available: www.ti.com/lit/pdf/slaa526
[38] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An energy-aware runtime for computational RFID," in *Proc. USENIX Symp. Networked Syst. Des. implementation*, 2011, pp. 197–210.
[39] K. Fischer, "Advancements in control system data authentication and verification," in *Proc. ASNE Intell. Ships Symp.*, 2017, pp. 25–25.
[40] A. Elmangoush, R. Steinke, T. Magedanz, A. A. Corici, A. Bourreau, and A. Al-Hezmi, "Application-derived communication protocol selection in M2M platforms for smart cities," in *Proc. Int. Conf. Intell. Next Gener. Netw.*, 2015, pp. 76–82.
[41] B. Buhrow, P. Riemer, M. Shea, B. Gilbert, and E. Daniel, "Block cipher speed and energy efficiency records on the MSP430: System design trade-offs for 16-bit embedded applications," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, 2014, pp. 104–123.
[42] D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov, "Triathlon of lightweight block ciphers for the internet of things," *J. Cryptographic Eng.*, vol. 9, no. 3, pp. 283–302, 2019.
[43] Texas Instruments Inc., "MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide," Texas Instrum. Inc., Dallas, TX, USA, 2014. Accessed: Oct. 4, 2022. [Online]. Available: www.ti.com/lit/pdf/slaa526
[44] R. Brown and K. Pier, "MSP430FRBoot – Main memory bootloader and over-the-air updates for MSP430™ FRAM large memory model," Dec. 2016, Accessed: Mar. 19, 2022. [Online]. Available: http://www.ti.com/lit/an/slaa721b/slaa721b.pdf
[45] T. Instruments, "MSP430 fram technology," Jun. 2014, Accessed: Mar. 19, 2022. [Online]. Available: https://www.ti.com/lit/an/slaa628b/slaa628b.pdf
[46] P.-A. Fouque, J. Jean, and T. Peyrin, "Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128," in *Proc. Annu. Cryptol. Conf.*, 2013, pp. 183–203.
[47] A. Biryukov and D. Khovratovich, "Related-key cryptanalysis of the full AES-192 and AES-256," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2009, pp. 1–18.
[48] B. Lee and J.-H. Lee, "Blockchain-based secure firmware update for embedded devices in an internet of things environment," *J. Supercomputing*, vol. 73, no. 3, pp. 1152–1167, 2017.
[49] L. Reynoso, "Mspboot – Main memory bootloader for MSP430 ™ microcontrollers," Jun. 2013, Accessed: Mar. 19, 2022. [Online]. Available: http://www.ti.com/lit/an/slaa600d/slaa600d.pdf
[50] S. Schulz *et al.*, "Boot attestation: Secure remote reporting with off-the-shelf IoT sensors," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2017, pp. 437–455.
[51] E. Dushku, M. M. Rabbani, M. Conti, L. V. Mancini, and S. Ranise, "SARA: Secure asynchronous remote attestation for IoT systems," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 3123–3136, 2020.
[52] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1198–1210, Sep. 2009.
[53] P. V. Nikitin, R. Martinez, S. Ramamurthy, H. Leland, G. Spiess, and K. Rao, "Phase based spatial identification of UHF RFID tags," in *Proc. IEEE Int. Conf. RFID*, 2010, pp. 102–109.
[54] J. Tan, "Robust downstream communication and storage for computational RFIDs," Dept. Softw. Technol., Delft Univ. Technol., Delft, Netherlands, 2015.
[55] Y. Su, A. Wickramasinghe, and D. C. Ranasinghe, "Investigating sensor data retrieval schemes for multi-sensor passive RFID tags," in *Proc. IEEE Int. Conf. RFID*, 2015, pp. 158–165.

Yang Su (Student Member, IEEE) received the BEng degree with first class Honours in electrical and electronic engineering from the University of Adelaide, Australia, in 2015. He worked as a research associate with the University of Adelaide from 2015-2016, and he is currently working toward the PhD degree. His research interests include hardware security, physical cryptography, embedded systems and computer security.

Michael Chesser received the BSc Advanced degree and the honours (First Class) degree in computer science from the University of Adelaide, Australia, in 2016 and in 2017, respectively. He has worked as a consultant with Chamonix Consulting and, more recently, with the School of Computer Science, University of Adelaide as a research associate. His research interests include compilers, embedded systems, system security and virtualization.

Yansong Gao received the MSc degree from the University of Electronic Science and Technology of China, in 2013, and PhD degree from the School of Electrical and Electronic Engineering, University of Adelaide, Australia, in 2017. He is now with the School of Computer Science and Engineering, NanJing University of Science and Technology, China. His current research interests include AI security and privacy, hardware security, and system security.

Alanson P. Sample received the PhD degree in electrical engineering from the University of Washington, in 2011. He is currently an Associate Professor with the Department of Electrical Engineering and Computer Science, University of Michigan. Prior to returning to academia, he spent the majority of his career working in academic minded industry research labs. Most recently Alanson was the executive lab director of Disney Research in Los Angeles. Prior to joining Disney, he worked as a research scientist with Intel Labs in Hillsboro. He also held a postdoctoral research position with the University of Washington. His research interests include broadly in the areas of human–computer interaction, wireless technology, and embedded systems.

Damith C. Ranasinghe received the PhD degree in electrical and electronic engineering from the University of Adelaide, Australia. In the past, he was a visiting scholar with the Massachusetts Institute of Technology, Cambridge, MA, and a postdoctoral research fellow with the University of Cambridge, Cambridge, UK. Currently, he is an associate professor with the University of Adelaide. His research interests include embedded systems, system security, autonomous systems, and deep learning.