cryptography

MDPI

Article

A Secure and Ultra-Lightweight Authentication Protocol for RFID Systems Using Epoch-Based Pseudonym Indexing

Pierre E. Abi-Char [1,*], Mehdi Al Housseini [2] and Mohammed Al-Husseini [1]

[1] College of Engineering and Technology, American University of the Middle East, Egaila 54200, Kuwait; mohammed.al-husseini@aum.edu.kw
[2] Beirut Research and Innovation Center, Lebanese Center for Studies and Research, Beirut 1234, Lebanon; mehdi.housseini@lebcsr.org
* Correspondence: pierre.abichar@aum.edu.kw

Abstract

Mobile Radio Frequency Identification (RFID) systems are emerging as a fundamental part of modern smart environments, enabling automatic identification, tracking, and data exchange among different mobile platforms. While these systems are increasingly being adopted, they have a major drawback: an RFID tag has very little computational power, and the wireless communication channels can be attacked by adversaries. Several authentication and key management mechanisms to protect data and provide secure access have been proposed to solve these problems. In this study, we propose a new scheme that improves system security through explicit three-party mutual authentication, epoch-based pseudonym indexing for $O(1)$ server lookup, and comprehensive resiliency against replay, impersonation, and man-in-the-middle attacks. An in-depth security analysis, along with performance evaluation, substantiates that the proposed protocol improves privacy and resilience without losing compatibility with low-cost RFID tags equipped only to perform lightweight cryptographic functions. This protocol also provides epoch-based unlinkability and is well suited for large-scale deployments, as found in healthcare, logistics, and Internet of Things (IoT) applications.

Keywords: RFID; authentication; tag; reader; data security; privacy; cryptography; scalability; Scyther; ProVerif

1. Introduction

Radio Frequency Identification (RFID) is a key technology in the Internet of Things (IoT) era. The importance of RFID is best demonstrated through its diverse and growing applications, including the Internet of Health Things (IoHT) [1], smart logistics target tracking [2], transportation systems, security and access control, smart agriculture, libraries and information management, IoT and wireless sensor networks, etc. In healthcare systems, RFID can provide various services that benefit medical staff and patients, including medication and asset management, patient safety, tracking and remote monitoring. In the latter, tags attached to patients, especially children, the elderly, or disoriented ones, store personal information for identification purposes, help medical staff quickly locate them, and remotely collect their physical health data. In smart logistics, RFID is used to improve monitoring accuracy and status visibility of tracked items within supply chain management. Often combined with sensor networks, RFID can enable both position and property tracking of mobile targets, which allows legal users to completely and visually master the cargo status, ensuring accurate amounts and appropriate conditions at specific sites [3].

Check for updates

Academic Editor: Jim Plusquellic

Received: 16 May 2026
Revised: 29 June 2026
Accepted: 9 July 2026
Published: 13 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

A standard RFID system consists of three basic components: a transponder (or RFID tag), a transceiver (or RFID reader), and a back-end database. Typically, an RFID reader contains a transceiver to modulate/demodulate RF signals that are exchanged with the tag, an antenna that transmits/receives such RF signals, and a controller that decodes the received data and connects with central information systems. The RFID tag comprises an integrated circuit (IC) and communicates with the RFID reader via an antenna. Tags can be classified as passive, active, or semi-passive. Passive tags do not come with an onboard power source, which makes them a good choice for high-volume use, as they are inexpensive. They derive their power from the electromagnetic (EM) field generated by the reader. Active tags have a power supply integrated into the system and are therefore capable of performing long-range reads and extended sensing functions. Each tag stores a unique identifier, user and application-specific data in the IC. Along with its antenna design and operating frequency, a tag's protocol compliance determines the tag's read range, data rate, and application compatibility. A back-end database (or host system) stores, processes and manages the data collected by RFID transceivers. It maps tag identifiers to corresponding business and operational data, performs authentication and integrity checks, and often interfaces with the management information systems of the enterprise [4].

The growing deployment and data transmission within RFID systems necessitate urgent attention to data security and privacy protection [5,6]. Communication channels in these systems are often wireless and insecure, making them vulnerable to numerous security threats. As a result, it is essential to authenticate the identities of all participating communication parties to achieve mutual trust [7], thus making the use of authentication protocols indispensable.

Security threats involve cloning, relay (man-in-the-middle) attacks, spoofing, replay attacks, tracking, denial-of-service (DoS) attacks, and eavesdropping [8,9]. A cloning attack occurs when an attacker illegally gains access by cloning a valid tag. In man-in-the-middle (MITM) attacks, an attacker places itself between a valid RFID tag and reader during communication establishment without either of them noticing, and can intercept, alter, forward, or inject messages, thereby controlling the communication flow. Spoofing is a technique employed by an adversary in which they imitate a valid RFID tag or reader by emitting false identifiers or responses that deceive the system into accepting fake information as genuine. In replay attacks, original messages exchanged between an RFID tag and a reader are recorded and replayed at a later point to impersonate the tag and perform actions without permission. With tracking attacks, an unauthorized third party can track the movement of an RFID tag and, as a result, trace either the object or person associated with it. In contrast, DoS attacks disrupt or completely block RFID operations by jamming the system with strong RF interference, or by overloading the tags or readers with excessive tag queries, rapidly repeated requests, or a large number of counterfeit tags. Lastly, eavesdropping is a passive security attack where an adversary intercepts data between a tag and a reader by capturing wireless signals without altering the data to obtain sensitive information, such as tag identifiers or authentication messages. These threats emphasize the importance of having robust authentication protocols specially designed for the resource-constrained RFID devices.

Authentication protocols in mobile RFID systems can be classified into three major categories, namely heavyweight, lightweight, and ultra-lightweight protocols [10]. Heavyweight protocols usually depend on symmetric and asymmetric cryptography (AES, RSA) or SHA-based hashes. In spite of their strong security and protection against replay, impersonation, and man-in-the-middle attacks, such protocols are computationally intensive and energy-consuming, which makes them unfit for passive tags. Lightweight protocols, on the other hand, adopt simplified cryptographic mechanisms, such as pseudorandom

number generators, cyclic redundancy checks, or simple lightweight hash functions to provide reasonable security while reducing the computational overhead. Finally, ultra-lightweight protocols use bitwise operations or simple permutations, which are suitable for the resource-limited passive or low-cost RFID tags. However, the level of security offered by these protocols is not as robust, and special care should be given to the design of the RFID system in this case to prevent vulnerabilities.

The remainder of this article is organized as follows: Section 2 reviews related work on RFID authentication protocols. Section 3 introduces our proposed authentication protocol, with a detailed specification of the system model, the epoch-based pseudonym mechanism, and the online authentication steps. Section 4 provides a comprehensive security analysis, including an informal analysis of attack resistance and formal verification. Section 5 evaluates the performance of the proposed protocol in terms of computational, communication, and storage overhead. Finally, Section 6 concludes the paper and discusses future work.

2. Related Work

Several studies on RFID authentication protocols have been reported in the literature over the past few years. In [11], the authors propose a lightweight, privacy- and data-confidentiality-preserving RFID protocol with an effective authentication mechanism for medical applications. Another lightweight RFID authentication scheme for healthcare systems is reported in [12]. Both quadratic residue methods and pseudorandom number generators are employed in this protocol to enhance the overall security of the parties. As per the authors' claim, the solution proposed therein maintains data confidentiality and provides resistance against the main threats affecting wireless mobile environments. A light and resilient authentication approach that is well suited for low-cost passive RFID tags is presented in [13]. This approach is based on the SAS-L2 framework and is specifically designed to meet minimal computational demands. In [14], the authors present a new secure and efficient lightweight authentication mechanism to mitigate the risks found in [12].

An ultra-lightweight authentication protocol that leverages word synthesis operations for data protection that achieves notable reductions in both communication and computational overheads is presented in [15]. The authors of [16] propose another ultra-lightweight authentication scheme for wireless mobile systems that minimizes computational overhead by using bit-permutation techniques. An ultra-lightweight mutual authentication scheme that combines rotation operations, T-functions, and timestamp mechanisms is reported in [17]. Herein, the authors employ tools such as Scyther and AVISPA to conduct formal security validations, where the results show resilience against a wide range of attacks, including replay-based attacks. In [18], an ultra-lightweight RFID authentication protocol named UDAP is proposed. It incorporates simple cryptographic techniques such as dot product, XOR, and left rotation, and enhances the levels of security and confidentiality while keeping the tag's computational load to a minimum. A performance analysis of the protocol is done using the Scyther simulation tool, which shows resistance to the most common security attacks.

An elliptic curve cryptography processor designed for passive ultra-high-frequency RFID tags is introduced in [19]. Sought for applications in anti-counterfeiting and banknote verification, it offers optimized performance, area and energy efficiency. In [20], a bidirectional authentication scheme for RFID systems is developed, where the design minimizes storage and computational overhead. Therein, the output of the hash function generated by the tag is divided into a segment that is assigned to reader validation and another for tag verification. A new improved RFID model that employs Chebyshev chaotic maps (CCMs) is proposed in [21] to address several limitations discovered in Akgun's scheme [22], such as vulnerability to impersonation, desynchronization, and secret leakage attacks.

In [23], an authentication framework for TMIS platforms providing both security and efficiency is presented. It ensures data confidentiality by combining quadratic residue computations and cryptographic hash operations, and is secure against replay attacks by relying on reader-generated timestamps. A secure and efficient mutual authentication scheme, referred to as SecMAP, is presented by Zhu in [24], which solves the system scalability and forward secrecy issues found in earlier work. A framework, called Au-Hota, that is resistant to impersonation and replay attacks and can operate with current ultra-high-frequency (UHF) RFID infrastructures, is proposed in [25]. The authors of [26] present an efficient group-based authentication mechanism that also incorporates privacy protection and prevents illicit tracking of tagged objects. In [27], an efficient and secure mutual authentication protocol for RFID systems that uses elliptic curve cryptography and can be applied for Internet of Vehicles (IoV) environments is proposed. To further improve the security requirements of IoV networks, a blockchain-driven framework is also provided therein.

The authors of [28] provide a comprehensive study of the RFID authentication protocols proposed in [11,29]. The study reveals several issues in these protocols, including susceptibility to desynchronization, impersonation, and replay attacks, in addition to weaknesses in preserving location privacy. Therefore, these authors proposed an authentication protocol to resolve these limitations, which was tested using BAN logic and the Scyther verification tool [28]. In [2], Xu et al. propose a three-party mobile RFID authentication protocol for smart logistics target tracking. The scheme is organized into an initial stage, an authentication stage, and an update stage; in the update stage, the pseudonym identifiers and keys are refreshed after each session. The protocol uses hash computations at the reader/cloud server side and exchange–cross (e.g., *Eac*) and rotation (e.g., *Rot*) bitwise operations at the tag side to keep tag computation low. To improve retrieval efficiency and reduce sensitive data disclosure risk on the cloud server, tag/reader records are stored as ciphertext in an index data table; however, the index value is still selected via an exhaustive search procedure, so the search time increases linearly with the number of tags. The protocol security is analyzed using BAN logic and further validated using the ProVerif tool and a random oracle model argument. In [30], a novel authentication protocol with strengthened security that incorporates pseudorandom number generation and vector dot product operations is introduced.

Nonetheless, many state-of-the-art lightweight protocols have one or more of the following drawbacks: (i) none of the existing works provides true three-party mutual authentication, where the server directly authenticates not only the reader but also the tag; (ii) several protocols, including those by Wang et al. [28] and Xu et al. [2], have $O(N)$ computational complexity on the server to identify tags and are therefore not scalable to large deployments; (iii) the use of shared secrets that are updated after each session may make the system vulnerable to desynchronization attacks, as protocol updates need to be carried out consistently at both sides; and (iv) user privacy cannot be ensured in these protocols due to static identification or predictable pseudonym generation mechanisms.

Among the protocols that do achieve $O(1)$ constant-time lookup, significant limitations remain. SecMAP [24] relies on quadratic-residue-based public key operations (modular squaring) on the tag side, requiring computational resources beyond the capability of standard passive EPC C1G2 tags. Sharma et al. [27] employs elliptic curve cryptography (ECC), which demands even heavier tag-side computation (scalar multiplication and point addition). Zhu et al. [14] achieve $O(1)$ lookup using hash-based indexing, but their security model does not incorporate explicit three-party mutual authentication with domain-separated tokens binding every authentication step to the session transcript. Furthermore, none of the existing $O(1)$ schemes provides epoch-based cross-session unlink-

ability, where tags automatically adopt fresh pseudonyms at each epoch without requiring per-session state updates. Table 1 provides a comparative summary of the key features across the most relevant related protocols and the proposed scheme. As shown in this table, the proposed protocol is the only scheme that simultaneously achieves explicit three-party mutual authentication, $O(1)$ constant-time server lookup, EPC C1G2 compatibility with only PRF operations on the tag, a fully stateless tag design immune to desynchronization, and epoch-based cross-epoch unlinkability.

Table 1. Feature comparison of related RFID authentication protocols.

Feature	SecMAP [24]	Sharma [27]	Wang [28]	Xu [2]	Zhu [14]	Alhasan [17]	Proposed
Three-party mutual auth.	Partial	N/A †	No	Partial	No	No	**Yes**
Server lookup	$O(1)$	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$\mathbf{O(1)}$
Tag-side primitive	MS + H	ECC	PRNG	Bitwise	PRNG	Bitwise	**PRF**
EPC C1G2 compat.	Marginal	No	Yes	Yes	Yes	Yes	**Yes**
Cross-epoch unlink.	No	No	No	Partial §	No	No	**Yes**
Stateless tag	No	No	No	No	No	No	**Yes**
Desync. resilient	Yes	—	Yes	Partial	Yes	Partial	**Yes**
Forward secrecy	Yes	Yes	No	No	No	No	**Yes**

† Sharma et al. do not report reader-side operations and does not target three-party mobile RFID settings. § Xu et al. update pseudonym identifiers after each session, providing some unlinkability but requiring stateful updates susceptible to desynchronization.

The contribution of this work is a novel protocol architecture that combines standard, well-understood building blocks—pseudorandom functions (PRFs), key derivation functions (KDFs), nonces, timestamps, and domain separation constants—with epoch-based pseudonym indexing in a way that simultaneously achieves the properties in Table 1. To our knowledge, no other existing protocol offers these properties together.

3. Proposed RFID Authentication Protocol

In this section, we present our optimized protocol for RFID authentication among a back-end server, a reader, and a tag. The protocol is designed for large-scale deployments with a high number of tags and focuses on two goals: (i) constant-time server processing per session and (ii) strong mutual authentication with explicit key and session confirmation. The notations used throughout this section are listed in Table 2. The overall message flow is shown in Table 3, while the main online steps are summarized in Table 4.

3.1. System Model and Notation

Each tag is manufactured with a permanent identifier TID. The server stores the set of all TID values and a master secret K_{master}, from which it derives per-tag keys

$$K_T = KDF(K_{\text{master}}, TID).$$

Each reader has a unique identifier RID and shares a symmetric key K_{RID} with the server for mutual authentication and policy enforcement. The protocol makes extensive use of a pseudorandom function PRF and an injective encoding function $\mathsf{encode}(\cdot)$ to bind the session transcript to all authentication tokens.

The encoding function is defined as:

$$\mathsf{encode}(x_1, x_2, \ldots, x_n) = \ell_1 \| x_1 \| \ell_2 \| x_2 \| \ldots \| \ell_n \| x_n$$

where ℓ_i is a 2-byte length prefix for field x_i and $\|$ denotes concatenation. This construction ensures injectivity where different input tuples always produce different encoded strings.

Table 2. Notation used in the proposed protocol.

Symbol	Description
TID, RID	Permanent identifiers of a tag and a reader
K_{master}	Master secret kept by the server
K_T	Per-tag key derived from K_{master} and TID
K_{RID}	Symmetric key shared by server and reader
N_r, N_t	Reader and tag nonces, respectively (64 bits each)
T_s	Server timestamp at session setup (32-bit Unix time)
ΔT	Maximum acceptable time window for timestamp validation (e.g., 300 s)
E	Epoch index (32 bits; e.g., 10 min time bucket)
$session\_id$	Unique identifier of the current session (64 bits)
PID	Epoch pseudonym of the tag (96 bits)
$EarlyAuth$	Early authentication token from the tag
Srv_OK	Initial server authentication token towards the reader
MR_1	Reader proof towards server under K_{RID}
$PROCEED$	Authorization token from server to reader
MR_2	Token from server to tag under K_T
$Auth_T$	Final tag liveness/authentication token
ACK	Final acknowledgment from server to reader
CONST_EA	Domain separation constant for the EarlyAuth token
CONST_MR1	Domain separation constant for the reader proof MR_1
CONST_PROCEED	Domain separation constant for the authorization token $PROCEED$
CONST_SRV_OK	Domain separation constant for the server token Srv_OK
CONST_MR2	Domain separation constant for the server–tag proof MR_2
CONST_AT	Domain separation constant for the final tag token $Auth_T$
CONST_ACK	Domain separation constant for the final acknowledgment ACK
"TIDL"	Tag ID label (epoch pseudonym label) and domain separation string used inside the PRF input when deriving PID
$hash(PID)$	Secure hashing function $hash(PID) = PID\%N$, where N is the number of tags
$MAP[E]$	Epoch hash table mapping $hash(PID) \rightarrow TID$ for epoch E
PRF	Pseudorandom function (keyed cryptographic primitive)
KDF	Key derivation function (e.g., HKDF or PBKDF2)
$\mathsf{encode}(\cdot)$	Injective encoder of transcript fields with length prefixes
$\mathsf{ROT_L}(X, Y)$	Bit rotate function to rotate X to the left by Y bits
$\mathsf{ROT_R}(X, Y)$	Bit rotate function to rotate X to the right by Y bits
N_{ST}	A 64-bit nonce shared between tag and server

Table 3. Message flow of the proposed RFID authentication protocol.

From	To	Message
Reader	Server	RID, N_r
Server	Reader	$session\_id, T_s, E, Srv\_OK$
Reader	Server	$MR_1, session\_id$
Server	Reader	$PROCEED, session\_id$
Reader	Tag	$session\_id, N_r, T_s, E$
Tag	Reader	$session\_id, N_t, HPID, EarlyAuth$
Reader	Server	$session\_id, N_t, HPID, EarlyAuth$
Server	Reader	$MR_2, session\_id$
Reader	Tag	MR_2
Tag	Reader	$Auth\_T, session\_id$
Reader	Server	$Auth\_T, session\_id$
Server	Reader	$ACK, session\_id$

Table 4. Main online steps in the proposed protocol.

Step	Main Transmission/Computation
1	Reader generates a fresh nonce N_r.
2	Reader $\rightarrow$ Server: $\{RID, N_r\}$.
3	The server records (RID, N_r), obtains T_s, derives the epoch E and a fresh $session\_id = \text{RAND}(64)$, and computes $Srv\_OK = PRF_{K_{RID}}(\text{encode}(\text{CONST\_SRV\_OK}, session\_id, N_r, T_s, E))$.
4	Server $\rightarrow$ Reader: $\{session\_id, T_s, E, Srv\_OK\}$.
5	Reader validates $\lvert T_{\text{current}} - T_s \rvert \leq \Delta T$, verifies Srv_OK, and computes MR_1.
6	Reader $\rightarrow$ Server: $\{MR_1, session\_id\}$
7	Server verifies MR_1 and computes $PROCEED$.
8	Server $\rightarrow$ Reader: $\{PROCEED, session\_id\}$
9	Reader verifies $PROCEED$.
10	Reader $\rightarrow$ Tag: $\{session\_id, N_r, T_s, E\}$
11	Tag generates N_t, computes $HPID$ and $EarlyAuth$ under K_T.
12	Tag $\rightarrow$ Reader: $\{session\_id, N_t, HPID, EarlyAuth\}$.
13	Reader $\rightarrow$ Server: $\{session\_id, N_t, HPID, EarlyAuth\}$.
14	Using $HPID$, the server computes PID. Next, computes $hash(PID)$ to locate candidate TID(s) in $MAP[E]/MAP[E-1]/MAP[E-2]$, verifies $EarlyAuth$ to identify the correct TID, and computes MR_2 under K_T.
15	Server $\rightarrow$ Reader: $\{MR_2, session\_id\}$
16	Reader $\rightarrow$ Tag: $\{MR_2\}$
17	Tag verifies MR_2, and computes $Auth\_T$.
18	Tag $\rightarrow$ Reader: $\{Auth\_T, session\_id\}$
19	Reader $\rightarrow$ Server: $\{Auth\_T, session\_id\}$.
20	Server verifies $Auth\_T$ and computes ACK under K_{RID}.
21	Server $\rightarrow$ Reader: $\{ACK, session\_id\}$.
22	Reader verifies ACK and finalizes the session.

Storage of Keys and Domain Separation Constants

In the setup phase, long-term keys and constants are distributed as follows: The domain separation constants are public fixed values that are hard-coded only in the parties that need to compute the corresponding tokens:

Server: Stores all TID, all K_T, K_{master}, N_{ST}, all per-reader keys K_{RID}, the epoch hash maps $MAP[E]$ (mapping each $hash(PID) \rightarrow TID$), and all domain separation constants: CONST_SRV_OK, CONST_MR1, CONST_EA, CONST_ACK, CONST_MR2, CONST_AT, and CONST_PROCEED.

Reader: Stores its identifier RID, the shared key K_{RID}, and the constants used in reader–server tokens: CONST_SRV_OK, CONST_MR1, CONST_PROCEED, and CONST_ACK.

Tag: Stores its identifier TID, its nonce N_{ST}, its pre-provisioned key K_T, the domain separation constants used in tag–server tokens (CONST_EA, CONST_MR2, CONST_AT), and the pseudonym label "TIDL" (4 bytes) used in PID derivation.

3.2. Epoch-Based Pseudonym Precomputation

The naive server receiving an anonymous tag message would need to read all stored TID values and recalculate the expected response with respect to each candidate tag, yielding a session-wise $O(N)$ cost, where N is the number of tags.

To prevent this, at the beginning of every epoch E (for example, once every ten minutes), the server performs an offline precomputation phase. It applies the respective per-tag key K_T for each stored TID and computes a pseudonym

$$PID = \text{Trunc}_{96}\big(\text{PRF}_{K_T}(\text{"TIDL"} \,\|\, E)\big),$$

truncated to 96 bits. In this, the literal ASCII string "TIDL" is used as a public domain separation label inside the PRF input so that epoch privacy-preserving pseudonyms cannot collide with any other PRF output generated under the same key K_T.

The mapping is saved as an entry in the epoch table:

$$MAP[E] \leftarrow (\text{hash}(PID) \rightarrow TID).$$

The epoch tables $MAP[E]$, $MAP[E-1]$, and $MAP[E-2]$ are organized in hash tables so as to support constant-time $O(1)$ lookup. Note that the hash function directly hashes the PID (which is 96 bits) to a memory address (bucket), and each bucket stores only the relevant TID value(s). During online authentication, the server calculates hash(PID) from a received pseudonym, accesses the candidate TID(s) stored in the bucket, and checks for the correct tag using EarlyAuth verification, as described in Step 14. In the common case, a bucket holds only one TID. In the unlikely case of hash collisions (different PIDs mapping to the same bucket) or truncation collisions (the same PID generated for different tags in an epoch), the bucket holds a small list of candidate TID values, all verified through EarlyAuth.

In order to handle clock drift and remain robust against epoch transitions, the server maintains three epoch tables in its memory: $MAP[E]$ (current epoch), $MAP[E-1]$ (previous epoch) and $MAP[E-2]$ (two epochs prior). When the system transitions to epoch $E+1$, the server:

1. Computes $MAP[E+1]$;
2. The newly added $MAP[E+1]$ becomes $MAP[E]$;
3. The old $MAP[E]$ is now $MAP[E-1]$;
4. The old $MAP[E-1]$ is now $MAP[E-2]$;
5. Finally, the server deletes old $MAP[E-2]$.

With a 10 min epoch, this three-epoch window accommodates up to about 30 min of clock skew, which would enable tags and readers with modest drift between their clocks to successfully authenticate. Figure 1 shows a seven-tag example for one epoch, say epoch E, where $hash(PID)$ will point to one of the addresses. Once the server extracts PID, it computes $hash(PID)$ and looks it up in the table for this epoch, called $MAP[E]$. For example, if $hash(PID)$ points to $Address_0$, the set of candidates C to be verified (Step 14 in Section 3.3) includes TID^1 and TID^4, and the remaining candidates will not be checked. If $hash(PID)$ points to $Address_2$, this means there is only one candidate to be verified, TID^5. If $hash(PID)$ points to $Address_3$, the server will have to look up the same $hash(PID)$ in the table for epoch $E-1$, called $MAP[E-1]$. If candidates are found, they will be verified; otherwise, the server will look up $MAP[E-2]$ in the table again. If no candidates are found, the server aborts. As a result, the server uses $hash(PID)$ to look up, in order, tables $MAP[E]$, $MAP[E-1]$, then $MAP[E-2]$, which allows for constant time identification for all cases, i.e., $O(1)$. This is very critical for scalability when the number of deployed tags is very large. Table 5 gives an example of the three epoch tables used for the constant-time PID resolution when there are no hash collisions, which is the best but a more common case.

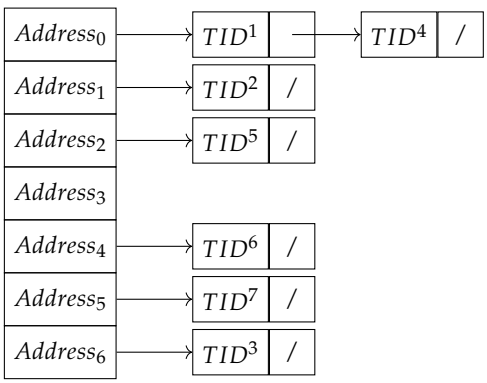

Figure 1. Example of server-side constant-time PID resolution for 1 epoch.

Table 5. Server-side constant-time PID resolution for the 3 epochs.

Epoch	Hash Table Entries			
Epoch (E)	$Address_0$	$Address_1$	…	$Address_{(N-1)}$
	TID^1	TID^2	…	TID^N
Epoch $(E-1)$	$Address_0$	$Address_2$	…	$Address_{(N-1)}$
	TID^1	TID^2	…	TID^N
Epoch $(E-2)$	$Address_0$	$Address_1$	…	$Address_{(N-1)}$
	TID^1	TID^2	…	TID^N

Notes: For each tag T^i with per-tag secret key K_T^i (derived from the server master key and the permanent identifier TID^i), the epoch pseudonyms are $PID_E^i = \text{Trunc}_{96}\Big(PRF_{K_T^i}(\text{“TIDL”} \| E)\Big)$. The ASCII string “TIDL” is a public domain separation label (Tag ID Label) used only inside the PRF input. Each bucket stores only the TID; the PID is not retained. During lookup, the server computes $hash(PID)$ from the received pseudonym and verifies the correct tag via EarlyAuth. Maintaining three epoch tables ensures successful authentication even with clock drift up to 30 min.

3.3. Online Authentication Protocol

The online phase consists of three parts: (i) mutual authentication between server and reader, (ii) early authentication and identification of the tag using its PID, and (iii) explicit mutual authentication between server and tag with final confirmation to the reader.

Server–reader authentication

Step 1: The reader generates a fresh 64-bit cryptographic nonce N_r to ensure freshness of the upcoming session.

Step 2: The reader sends the pair $\{RID, N_r\}$ to the server, where RID is the reader identifier and N_r is the freshly generated nonce.

Step 3: Upon receiving the request, the server records (RID, N_r), samples the current timestamp T_s (Unix time in seconds), derives the epoch index $E = \lfloor T_s/600 \rfloor$ (assuming 10 min epochs), and creates a unique *session_id* as a 64-bit cryptographically secure random value: $session\_id = \text{RAND}(64)$. The server then computes an initial authentication token

$$Srv\_OK = PRF_{K_{RID}}\big(\text{encode}(\text{CONST\_SRV\_OK}, session\_id, N_r, T_s, E)\big). \tag{1}$$

Step 4: The server sends $\{session\_id, T_s, E, Srv\_OK\}$ to the reader so that both parties share the session parameters and the reader can authenticate the server.

Step 5: The reader first validates the timestamp freshness by checking $|T_{\text{current}} - T_s| \leq \Delta T$, where T_{current} is the reader's current time and ΔT is the maximum acceptable time window (e.g., 300 s). If the timestamp is outside this window, the protocol aborts. Otherwise, the reader verifies Srv_OK by recomputing it and doing the following comparison:

$$PRF_{K_{RID}}\big(\text{encode}(\text{CONST\_SRV\_OK}, session\_id, N_r, T_s, E)\big) \stackrel{?}{=} Srv\_OK. \tag{2}$$

If the check fails, the protocol aborts. Otherwise, the reader proves its own legitimacy towards the server by computing

$$MR_1 = PRF_{K_{RID}}\big(\text{encode}(\text{CONST\_MR1}, session\_id, N_r, T_s, E)\big) \tag{3}$$

under K_{RID}.

Step 6: The reader sends $\{MR_1, session\_id\}$ to the server.

Step 7: The server verifies MR_1 by recomputing Equation (3) and comparing the received MR_1 to the newly computed one. If the equality check fails, it aborts the protocol. Otherwise, it computes the following authorization token:

$$PROCEED = PRF_{K_{RID}}\big(\text{encode}(\text{CONST\_PROCEED}, session\_id, N_r, T_s, E)\big) \tag{4}$$

This token binds the reader's authorization to the current session.
Step 8: The server sends $\{PROCEED, session\_id\}$ to the reader.
Step 9: The reader verifies $PROCEED$ by recomputing and comparing. If the check fails, the protocol aborts. Otherwise, it proceeds to interact with the tag.

Early tag authentication and identification

Step 10: After successfully verifying $PROCEED$, the reader sends $\{session\_id, N_r, T_s, E\}$ to the tag so that the tag can join the same authenticated session.
Step 11: Upon reception, the tag generates a fresh 64-bit nonce N_t and computes $EarlyAuth$, PID, and $HPID$ as follow:

$$\begin{aligned} EarlyAuth &= PRF_{K_T}(\text{encode}(\text{CONST\_EA}, session\_id, N_r, T_s, N_t)), \\ PID &= \text{Trunc}_{96}\big(PRF_{K_T}(\text{“TIDL”} \,\|\, E)\big), \\ HPID &= PID \oplus ROT_L((N_t||T_S), N_{ST}) \end{aligned} \tag{5}$$

Step 12: The tag sends $\{session\_id, N_t, HPID, EarlyAuth\}$ back to the reader.
Step 13: The reader forwards $\{session\_id, N_t, HPID, EarlyAuth\}$ to the server so that the server can identify and authenticate the tag.

Explicit server–tag mutual authentication

Step 14 (PID lookup, EarlyAuth verification, and MR_2 computation): Upon receiving the message from the reader, the server computes: $PID = HPID \oplus ROT_R((N_t||T_S), N_{ST})$. Next, the server performs constant-time lookups in the epoch tables. For each table, the server computes $hash(PID)$ from the received pseudonym to locate the bucket, and retrieves the candidate TID value(s) stored there:

1. First attempts epoch E;
2. If not found, attempts epoch $E-1$;
3. If still not found, attempts epoch $E-2$.

Let C be the set of candidate TID values that are returned on the first successful lookup. When three lookups fail ($C = \emptyset$), the server aborts the protocol as the tag is unknown or the epoch is outside the maximum synchronization window. In case the candidate is a singleton $|C| = 1$, the bucket contains only one candidate. Confirming whether this candidate is the correct tag is done through EarlyAuth verification below. In the unlikely case of a collision ($|C| > 1$)—whether due to hash collisions (multiple PIDs mapping to one bucket) or truncation collisions (similar tags produce the same 96-bit PID)—the server loops through all candidates: for each $TID \in C$, it retrieves the corresponding key K_T and $CONST\_EA$ and generates the token $PRF_{K_T}\big(\text{encode}(\text{CONST\_EA}, session\_id, N_r, T_s, N_t)\big)$, and checks if it is equal to $EarlyAuth$:

$$PRF_{K_T}\big(\text{encode}(\text{CONST\_EA}, session\_id, N_r, T_s, N_t)\big) \stackrel{?}{=} EarlyAuth. \tag{6}$$

The server tries each candidate in C until one produces a matching $EarlyAuth$. If no candidate matches, the protocol aborts (this indicates either an attack or corruption). Otherwise, the server has successfully identified the tag and binds the session to it by storing the association:

$$(session\_id \rightarrow TID, RID, N_r, N_t, T_s, E).$$

The server then computes

$$MR_2 = PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_MR2}, session\_id, N_r, T_s, N_t)\big). \tag{7}$$

Step 15: The server sends $\{MR_2, session\_id\}$ to the reader.
Step 16: The reader sends $\{MR_2\}$ to the tag unchanged, simply serving as a relay link between the server and the tag.
Step 17: The tag verifies

$$PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_MR2}, session\_id, N_r, T_s, N_t)\big) \stackrel{?}{=} MR_2. \tag{8}$$

If the equality holds, the tag authenticates the server for this exact session and computes a final liveness and authentication token

$$Auth\_T = PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_AT}, session\_id, N_r, T_s, N_t)\big). \tag{9}$$

Step 18: The tag sends $\{Auth\_T, session\_id\}$ back to the reader.
Step 19: The reader forwards $\{Auth\_T, session\_id\}$ to the server so that the server can complete tag authentication.
Step 20: The server recomputes the expected value of $Auth\_T$ from the stored session state (retrieved using $session\_id$). If the check succeeds, it computes a final acknowledgment

$$ACK = PRF_{K_{RID}}\big(\mathsf{encode}(\text{CONST\_ACK}, session\_id, N_r, N_t, T_s)\big) \tag{10}$$

under K_{RID}.
Step 21: The server sends $\{ACK, session\_id\}$ to the reader.
Step 22: The reader verifies ACK by recomputing and comparing. If valid, the reader marks the tag as successfully authenticated and the session as complete.

4. Security Analysis of the Proposed Protocol

4.1. Informal Analysis

In this section, we analyze the security of the protocol described in Section 3. We describe the attacker model and then examine resistance to common attacks, including replay, impersonation, man-in-the-middle (MITM), desynchronization, and traceability.

To analyze the security threats to the proposed model, we assume a Dolev–Yao adversary with full control over the communication channel. In particular, this adversary can eavesdrop on all messages, intercept them, modify their contents, inject new messages of their choice, or replay previously recorded traffic. However, we assume that the underlying cryptographic primitives are secure: the pseudorandom function PRF behaves as a standard PRF, the key derivation function KDF is secure, and the encoding function $\mathsf{encode}(\cdot)$ is injective. In other words, without knowing the relevant secret key (either K_T or K_{RID}), the adversary cannot predict or invert PRF outputs, nor can it find two different transcripts that produce the same encoded string.

4.1.1. Resistance to Replay Attacks

Replay attacks are mitigated by combining fresh nonces, unique session identifiers, and timestamp/epoch values in every execution of the protocol. Each session uses a fresh $session\_id$ that the server provides in Step 3, including both a reader nonce N_r and a tag nonce N_t generated in Steps 1 and 11, respectively. These values ensure freshness so that messages from an old session cannot be used in a new one without detection.

The reader samples a new random challenge N_r at the start of each execution, and the tag responds with its own new N_t; both nonces are fed to the cryptographic tokens. All authentication tokens—Srv_OK, MR_1, $EarlyAuth$, MR_2, $Auth\_T$, and ACK—are PRF evaluations over $\mathsf{encode}(\cdot)$ inputs that incorporate freshness values, specifically the nonce(s), $session\_id$, timestamp T_s, and epoch index E. As a result, replaying any message from an earlier session produces a PRF input that does not match the current session context, causing the receiving party's validation to fail.

Step 5 validates the timestamp $|T_{\text{current}} - T_s| \leq \Delta T$ before accepting any server messages; therefore, we are protected with one more layer. This, in turn, prevents replay of old sessions which are already outside the permissible time window ΔT (usually 300 s). An attacker who manages to capture a full transcript of a session and replay it gets the problem that, within this interval, replaying the values for N_r and N_t will no longer agree, since they have fresh ones in the session being replayed; thus, all PRF verifications will fail.

A further layer of protection is the use of domain separation constants (e.g., CONST_SRV_OK, CONST_MR1, CONST_EA). Tokens are all associated with a specific constant and therefore with a concrete step of the protocol. This prevents an attacker from being able to carry a valid token through phases (i.e., an old $Auth\_T$) and replay it in an alternate context. Even when the opponent has full pre-authentication transcripts, replaying those messages does not yield a valid session, given that the reader and server anticipate fresh values; the final ACK provides an explicit confirmation that the session is new and complete.

In conclusion, using fresh nonces, session identifiers that uniquely identify consecutive sessions, timestamps, and PRF evaluation across domain-separated inputs provides high resistance to replay attacks.

4.1.2. Resistance to Impersonation Attacks

Tag Impersonation

To masquerade as a legitimate tag, an adversary must generate responses that the server accepts as coming from a tag with key K_T. The critical tag-side values are $EarlyAuth$, $HPID$, PID, and $Auth\_T$, all of which are computed under K_T:

$$\begin{aligned} EarlyAuth &= PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_EA}, session\_id, N_r, T_s, N_t)\big), \\ PID &= \text{Trunc}_{96}\big(PRF_{K_T}(\text{“TIDL”} \,\|\, E)\big), \\ Auth\_T &= PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_AT}, session\_id, N_r, T_s, N_t)\big). \end{aligned}$$

Utilizing K_T and the transcript received in Steps 10–13, the server verifies $EarlyAuth$ at Step 14 and $Auth\_T$ at Step 20. Since both tokens are associated with PRF computations that rely on the appropriate key, an adversary cannot generate matching outputs. A failing proof is instantly aborted by the server. Thus, an adversary can never convince the server that some arbitrary device is a given tag without having somehow acquired the tag's secret key K_T.

Reader Impersonation

Impersonating a legitimate reader similarly requires knowledge of the K_{RID} key shared between the reader and the server. During the initial handshake, the server first authenticates itself by sending Srv_OK, and then, the reader proves its identity using:

$$MR_1 = PRF_{K_{RID}}\big(\mathsf{encode}(\text{CONST\_MR1}, session\_id, N_r, T_s, E)\big)$$

In that scenario, if the attacker tries to impersonate the reader, it will need to find MR_1 using $(session\_id, N_r, T_s, E)$, which is computationally infeasible without K_{RID}. In Step 7,

the server aborts if there is a mismatch in MR_1. The server also sends $PROCEED$ as an authorization token under K_{RID}, which is authenticated by the reader before continuing the interaction with the tag. A fraudulent reader, unable to produce valid Srv_OK and $PROCEED$ tokens without knowing K_{RID}, cannot complete a protocol run. Hence, an adversary cannot pose as a valid reader to the server, and it cannot postpone successful reader authorization information for upcoming executions.

Server Impersonation

Impersonating the back-end server towards either the reader or the tag is also infeasible under the assumed cryptographic hardness. A fake server that replies to the reader's initial message would need to generate a valid token without knowing K_{RID} using:

$$Srv\_OK = PRF_{K_{RID}}\big(\mathsf{encode}(\text{CONST\_SRV\_OK}, session\_id, N_r, T_s, E)\big)$$

The reader checks this value in Step 5, and any invalid token leads to an immediate abort.

Likewise, in the tag phase, the key message from server to tag is MR_2:

$$MR_2 = PRF_{K_T}\big(\mathsf{encode}(\text{CONST\_MR2}, session\_id, N_r, T_s, N_t)\big)$$

The legitimate tag verifies this value in Step 17. An attacker who does not know K_T cannot construct MR_2 that passes the tag's check, and the tag will refuse to continue the session.

4.1.3. Resistance to Man-in-the-Middle Attacks

In the Dolev–Yao model, a man-in-the-middle adversary operates as an active attacker who intercepts in-flight messages and modifies them with the purpose of deceiving some or all parties involved. Our proposed protocol is robust against such manipulation, as every security-critical field is cryptographically bound to the others via PRF and $\mathsf{encode}(\cdot)$. If a man-in-the-middle attack changes some part of a message—*session_id*, nonce, token, etc.—the corresponding PRF verification at the receiving side fails. The values received by the tag, namely $(session\_id, N_r, T_s, E)$ in Step 10, are then used as input variables to the $EarlyAuth$ and MR_2 computations on the server and the tag, respectively. Any change in these values in transit means that the recalculated PRF values will not be equal to the received token, so the protocol aborts. Likewise, corrupting the tag's response at Step 12 (e.g., modifying $HPID$ or $EarlyAuth$ and forwarding its message to the server) results in either a failed lookup in the epoch tables or a failed verification of $EarlyAuth$ at Step 14. Each token (Srv_OK, MR_1, $EarlyAuth$, MR_2, $Auth\_T$, and ACK) is a PRF output that applies a unique domain separation constant and the values of all relevant fields, so an attacker cannot modify just some subset of the fields without failing equality checks. This helps keep MITM resistance for the final exchange of $Auth\_T$ and ACK. The tag continues to accept the server only after MR_2 is verified, and the reader considers a session successful only after ACK from the server is verified. Specifically, the ACK token computed over $(session\_id, N_r, N_t, T_s)$ by K_{RID} ensures that the server has received and verified the tag's contribution. Since an attacker would have to attempt to splice together partial transcripts, this three-party confirmation complicates matters because any such splicing will ultimately fail one of the PRF checks and be detected.

4.1.4. Resistance to Desynchronization Attacks

Desynchronization attacks attempt to force the server and tag into inconsistent internal states so that legitimate future authentications fail. This is a common issue for protocols

that update long-term secrets or indexes after each run. Our protocol is inherently robust against such attacks because neither the tag nor the server maintains a per-session evolving secret that must be incrementally synchronized. The tag's main long-term secret is K_T, which remains static and is derived from the master key and TID. The pseudonym PID is computed as a function of K_T and the epoch index E:

$$PID = \text{Trunc}_{96}\big(PRF_{K_T}(\text{“TIDL”} \,\|\, E)\big).$$

This value is not stored as a mutable state; it is recomputed as needed. The server similarly does not update any per-tag secret in an irreversible way during the session. If an adversary blocks, delays, or modifies messages, the worst consequence is an aborted session. In the next run, the tag and the server simply perform the protocol again using the same K_T and the current epoch index, and they successfully authenticate.

The epoch-based pseudonym system is specifically designed to tolerate timing mismatches and prevent desynchronization. The server maintains three concurrent epoch tables ($MAP[E]$, $MAP[E-1]$, $MAP[E-2]$), providing approximately 30 min of tolerance for clock drift (assuming 10 min epochs). During epoch transitions, both the old and new pseudonyms remain valid simultaneously, ensuring smooth authentication even if the reader and tag have slightly different time synchronization.

If an attacker attempts to cause desynchronization by blocking messages during an epoch transition, the protocol remains resilient:

- If the tag uses epoch E but the server has advanced to $E+1$, the lookup in $MAP[E]$ (now in the previous epoch slot) still succeeds.
- If clock drift causes the tag to lag by up to two epochs, $MAP[E-2]$ provides recovery.
- No permanent state corruption occurs—both parties continue using their static keys.

An attacker cannot exploit this mechanism to cause permanent desynchronization: at most, feeding an incorrect epoch E to the tag will cause that particular session to fail, but both sides will remain synchronized in terms of their long-term secrets and will be able to authenticate again in the correct epoch. There is no monotonically increasing counter that can be unintentionally incremented beyond recovery.

As a result, the protocol avoids the class of desynchronization failures seen in earlier ultra-lightweight RFID protocols that rely on stateful key or index updates after each session.

4.1.5. Tag Privacy, Traceability, and Linkability Attacks

We achieve privacy by using epoch-based pseudonyms rather than static identifiers. Instead of broadcasting TID out in the open, the tag computes and sends the term $HPID$:

$$HPID = PID \oplus ROT_L((N_t||T_S), N_{ST})$$

where

$$PID = \text{Trunc}_{96}\big(PRF_{K_T}(\text{“TIDL”} \,\|\, E)\big),$$

In fact, $HPID$ varies for every session within every epoch E as new N_t and T_S are generated every session by any tag. Therefore, and on longer time scales, an observer who is trying to track a tag in the same epoch or in different epochs only sees uncorrelated pseudonyms and thus cannot link those sessions without knowing K_T, N_{ST}, and $TIDL$. It is worthy to note that in our protocol, K_T, N_{ST} and $TIDL$ are never sent in plaintext. This is why it is not possible to track a tag in the long term solely based on its identifiers. Additionally, the fresh N_t and distinct PRF outputs ($EarlyAuth$, $Auth\_T$) in each authentication transcript for an epoch further ensure that even repeated runs are not bitwise identical. This makes it impossible for an attacker to correlate transcripts using repeating patterns. Moreover,

for a single epoch window of duration Δ_{epoch} (e.g., 10 min), the pseudonym PID stays constant for a given tag, but $HPID$ is not constant as it varies each session. An attacker who is trying to query or observe a tag multiple times in the same epoch will see different $HPID$ and cannot determine whether these readings originate from which physical tag or if they belong to the same tag. However, on the server side, the server can still derive the same stable PID within an epoch, which allows the server to maintain the epoch-indexed hash map and perform constant-time $O(1)$ tag lookup. This is very essential for scalability when millions of tags are deployed.

4.1.6. Forward Secrecy

Forward secrecy is an important security property that ensures previously exchanged messages must be protected and should not be traced by an adversary if the tag's current secret credentials are compromised. In the proposed protocol, tag secrets $TIDL$, K_T, and N_{ST} are never transmitted in cleartext. Assuming the tag is not physically compromised, and within one epoch, an adversary reading the current session's $HPID$ cannot recover K_T and $TIDL$ and all previous $HPID$, since all $HPID$ are generated using new random challenges N_{ST}, T_S, and N_t for each session. In addition, the adversary can not recover the previous $HPID$ from previous sessions, including Srv_OK, MR_1, MR_2, $PROCCED$, and $EarlyAuth$, as these terms are generated using new random challenges for each session. However, assume that an adversary was able to obtain the tag's secrets, $TIDL$, K_T, and N_{ST}, in the current session information. The adversary cannot compute the PID for all past epochs for this tag as PID was never sent in the clear, and all previous $HPID$ were generated using new random challenges N_{ST}, T_S, and N_t. In addition, the adversary cannot compute $EarlyAuth$ and $Auth_T$, as these terms are generated using new random challenges N_r, T_S, and N_t for each epoch and each session. In addition, the use of PRF by our protocol guarantees that none of the transmitted messages, including Srv_OK, $MR1$, $MR2$, $PROCCED$, and $EarlyAuth$, can reveal any possible information to derive any of the previous $HPID$, $EarlyAuth$, and $Auth_T$. Therefore, an adversary cannot recover or trace past communications.

In addition, the tag's secrets ($TIDL$, K_T, N_{ST}) are never exposed during any session, and they are assumed to reside in tamper-resistant hardware where an adversary cannot easily extract the protected data from the tag's internal memory through physical tampering. Secret extraction through physical compromise of the tag is a standard hardware security assumption in RFID deployments and is assumed to be outside the scope of this cryptographic protocol. As a solution, and upon detecting or receiving a report of a compromised tag, we assume that the server initiates a tag revocation procedure to prevent further unauthorized access.

4.2. Formal Security Analysis

To complement the informal analysis, we provide formal verification of the protocol's security properties using the Scyther and ProVerif verification tools.

4.2.1. Scyther Verification

We modeled the protocol in the Scyther automated verification tool [31] and verified it under the Dolev–Yao attacker model. The protocol was specified in Scyther's Security Protocol Description Language (SPDL), modeling the three-party communication among Server (S), Reader (R), and Tag (T) with symmetric keys $k(S, R)$, representing K_{RID}, and $k(S, T)$, representing K_T. The epoch derivation $E = \lfloor T_s/600 \rfloor$ is modeled as a public hash function `epoch` applied to T_s, capturing the deterministic relationship between the timestamp and epoch index.

The verification was conducted with the following parameters:

- Maximum number of runs: Three.
- Matching type: Typed matching.
- Search pruning: Find the best attack.

Table 6 summarizes the Scyther verification results. All security claims were verified with status "Ok—No attacks within bounds."

Table 6. Scyther verification results for the proposed protocol.

Role	Claim	Property	Status
R	R2	Commit S, sid, N_r, T_s, E	Ok
R	R3	Secret $k(S, R)$	Ok
S	S2	Commit R, sid, N_r, T_s, E	Ok
S	S4	Commit T, sid, N_r, T_s, N_t	Ok
S	S5	Secret $k(S, R)$	Ok
S	S6	Secret $k(S, T)$	Ok
T	T2	Commit S, sid, N_r, T_s, N_t	Ok
T	T3	Secret $k(S, T)$	Ok

The verification confirmed the following security properties:

- **Secrecy**: The symmetric keys K_{RID} (modeled as $k(S, R)$) and K_T (modeled as $k(S, T)$) remain confidential. No attack was found that could reveal these keys to the adversary.
- **Mutual Authentication**: All three parties achieve authentication through the commit claims:
  - Reader authenticates Server (R2: Commit S, sid, N_r, T_s, E).
  - Server authenticates Reader (S2: Commit R, sid, N_r, T_s, E).
  - Server authenticates Tag (S4: Commit T, sid, N_r, T_s, N_t).
  - Tag authenticates Server (T2: Commit S, sid, N_r, T_s, N_t).
- **Agreement:** The commitments verifying that all parties agree on the session parameters provide assurance against man-in-the-middle attacks.

The analysis with Scyther found no attacks within its bounded search space (up to three concurrent runs), which provides strong evidence that the protocol meets the claimed secrecy and authentication properties under the Dolev–Yao attacker model. We note that Scyther performs bounded model checking; therefore, these results do not constitute a full mathematical proof but rather a rigorous automated verification within the explored state space.

Several inherent modeling limitations should be noted. First, Scyther cannot reason about time, so the timestamp freshness check ($|T_{\text{current}} - T_s| \leq \Delta T$) is not captured; freshness relies entirely on the nonce-based mechanisms in the model. Second, the epoch index E is modeled as a deterministic function of T_s via a public hash function; however, the model treats each session's T_s as unique, so within-epoch pseudonym reuse (the deliberate privacy trade-off discussed in Section 4) is not exercised by Scyther. Third, since Scyther parametrizes the protocol with fixed role identities, the server directly pattern matches the tag's key $k(S, T)$, and the epoch table lookup mechanism (hash-based PID resolution with collision handling) is not formally verified; its correctness is argued informally in Section 3.2. Complementary unbounded verification using ProVerif is presented in Section 4.2.2. Listing 1 presents the Scyther SPDL specification of the proposed protocol.

Listing 1. Scyther SPDL specification with Running/Commit claim pairs.

```
/*
* Scyther Verification Model
* Protocol: Ultra-lightweight RFID Authentication Protocol
* With proper Running/Commit claim pairs for authentication
*/
hashfunction PRF;
hashfunction PIDGEN;
hashfunction epoch; // models E = floor(Ts/600)
const CSRVOK; const CMR1; const CPROCEED; const CEA; const CMR2; const CAT; const CACK; const CTIDL;

protocol RFIDAuth(S, R, T)
{
  /* READER ROLE */
  role R
  {
    fresh Nr: Nonce;
    var sid: Nonce; var Ts: Nonce; var E: Ticket; var Nt: Nonce;
    var tagPID: Ticket; var earlyAuth: Ticket; var mr2Token: Ticket; var authT: Ticket;

    send_1(R, S, Nr);
    recv_2(S, R, sid, Ts, E, PRF(k(S,R), CSRVOK, sid, Nr, Ts, E));
    claim(R, Running, S, sid, Nr, Ts, E);                    /* Reader starts session */
    send_3(R, S, PRF(k(S,R), CMR1, sid, Nr, Ts, E), sid);
    recv_4(S, R, PRF(k(S,R), CPROCEED, sid, Nr, Ts, E), sid);
    send_5(R, T, sid, Nr, Ts, E);
    recv_6(T, R, sid, Nt, tagPID, earlyAuth);
    send_7(R, S, sid, Nt, tagPID, earlyAuth);
    recv_8(S, R, mr2Token, sid);
    send_9(R, T, mr2Token);
    recv_10(T, R, authT, sid);
    send_11(R, S, authT, sid);
    recv_12(S, R, PRF(k(S,R), CACK, sid, Nr, Nt, Ts), sid);
    claim(R, Commit, S, sid, Nr, Ts, E);                     /* Reader authenticates Server */
    claim(R, Secret, k(S,R));
  }

  /* SERVER ROLE */
  role S
  {
    fresh sid: Nonce; fresh Ts: Nonce;
    var Nr: Nonce; var Nt: Nonce;

    recv_1(R, S, Nr);
    claim(S, Running, R, sid, Nr, Ts, epoch(Ts));            /* Server starts with Reader */
    send_2(S, R, sid, Ts, epoch(Ts), PRF(k(S,R), CSRVOK, sid, Nr, Ts, epoch(Ts)));
    recv_3(R, S, PRF(k(S,R), CMR1, sid, Nr, Ts, epoch(Ts)), sid);
    claim(S, Commit, R, sid, Nr, Ts, epoch(Ts));             /* Server authenticates Reader */
    send_4(S, R, PRF(k(S,R), CPROCEED, sid, Nr, Ts, epoch(Ts)), sid);
    recv_7(R, S, sid, Nt, PIDGEN(k(S,T), CTIDL, epoch(Ts)), PRF(k(S,T), CEA, sid, Nr, Ts, Nt));
    claim(S, Running, T, sid, Nr, Ts, Nt);                   /* Server starts with Tag */
    send_8(S, R, PRF(k(S,T), CMR2, sid, Nr, Ts, Nt), sid);
    recv_11(R, S, PRF(k(S,T), CAT, sid, Nr, Ts, Nt), sid);
    claim(S, Commit, T, sid, Nr, Ts, Nt);                    /* Server authenticates Tag */
    send_12(S, R, PRF(k(S,R), CACK, sid, Nr, Nt, Ts), sid);
    claim(S, Secret, k(S,R));
    claim(S, Secret, k(S,T));
  }

  /* TAG ROLE */
  role T
  {
    fresh Nt: Nonce;
    var sid: Nonce; var Nr: Nonce; var Ts: Nonce; var E: Ticket;

    recv_5(R, T, sid, Nr, Ts, E);
    claim(T, Running, S, sid, Nr, Ts, Nt);                   /* Tag starts session */
    send_6(T, R, sid, Nt, PIDGEN(k(S,T), CTIDL, E), PRF(k(S,T), CEA, sid, Nr, Ts, Nt));
    recv_9(R, T, PRF(k(S,T), CMR2, sid, Nr, Ts, Nt));
    claim(T, Commit, S, sid, Nr, Ts, Nt);                    /* Tag authenticates Server */
    send_10(T, R, PRF(k(S,T), CAT, sid, Nr, Ts, Nt), sid);
    claim(T, Secret, k(S,T));
  }
}
```

4.2.2. ProVerif Verification

To complement the bounded Scyther verification, we performed unbounded formal verification using the ProVerif automated security protocol verifier [32]. Unlike Scyther, which explores a bounded number of protocol runs, ProVerif analyzes security properties for an unbounded number of concurrent sessions under the Dolev–Yao attacker model using resolution-based techniques on Horn clauses.

The protocol was modeled in ProVerif's applied pi-calculus language, with the PRF modeled as a symbolic unary function, the KDF as a key derivation constructor, and all communication occurring over a public channel fully accessible to the adversary. The three protocol roles (server, reader, tag) were each modeled with unbounded replication. Table 7 summarizes the verification results. Listing 2 presents the ProVerif source code of the proposed protocol.

Listing 2. ProVerif Verification Model.

```
(* ============================================================ *)
(* ProVerif Verification Model                                  *)
(* Protocol: Ultra-lightweight RFID Authentication Protocol *)
(*          with Epoch-based Pseudonym Indexing                 *)
(* Verifies: Secrecy, Mutual Authentication, Agreement    *)
(* Attacker model: Dolev--Yao (unbounded sessions)         *)
(* ============================================================ *)

(* --- Channel --- *)
free ch: channel.

(* --- Cryptographic primitives --- *)
fun PRF(bitstring, bitstring): bitstring.
fun KDF(bitstring, bitstring): bitstring.
fun PIDGEN(bitstring, bitstring): bitstring.

(* --- Domain separation constants (public) --- *)
free CONST_SRVOK: bitstring.
free CONST_MR1:  bitstring.
free CONST_PROCEED:bitstring.
free CONST_EA:   bitstring.
free CONST_MR2:  bitstring.
free CONST_AT:   bitstring.
free CONST_ACK:  bitstring.

(* --- Secret values --- *)
free KRID:   bitstring [private].
free KT:     bitstring [private].
free TID_val: bitstring [private].

(* ============================================================ *)
(* Secrecy queries                                              *)
(* ============================================================ *)
query attacker(KRID).
query attacker(KT).
query attacker(TID_val).

(* ============================================================ *)
(* Authentication events                                        *)
(* ============================================================ *)
event serverStartsReader(bitstring, bitstring, bitstring, bitstring).
event readerAuthServer(bitstring, bitstring, bitstring, bitstring).
event tagStarts(bitstring, bitstring, bitstring, bitstring).
event serverAuthTag(bitstring, bitstring, bitstring, bitstring).
event tagAuthServer(bitstring, bitstring, bitstring, bitstring).
event serverComplete(bitstring, bitstring, bitstring, bitstring).

(* --- Reader authenticates Server --- *)
query sid: bitstring, nr: bitstring, ts: bitstring, e: bitstring;
  event(readerAuthServer(sid, nr, ts, e))
  ==> event(serverStartsReader(sid, nr, ts, e)).

(* --- Server authenticates Tag --- *)
query sid: bitstring, nr: bitstring, ts: bitstring, nt: bitstring;
```

```
  event(serverAuthTag(sid, nr, ts, nt))
  ==> event(tagStarts(sid, nr, ts, nt)).

(* --- Tag authenticates Server --- *)
query sid: bitstring, nr: bitstring, ts: bitstring, nt: bitstring;
  event(tagAuthServer(sid, nr, ts, nt))
  ==> event(serverAuthTag(sid, nr, ts, nt)).

(* --- Full session agreement --- *)
query sid: bitstring, nr: bitstring, ts: bitstring, nt: bitstring;
  event(serverComplete(sid, nr, ts, nt))
  ==> event(tagAuthServer(sid, nr, ts, nt)).

(* ============================================================ *)
(* READER PROCESS                                               *)
(* ============================================================ *)
let processReader() =
  new Nr: bitstring;
  (* Step 2: R -> S : Nr *)
  out(ch, Nr);
  (* Step 4: S -> R : sid, Ts, E, Srv_OK *)
  in(ch, (sid: bitstring, Ts: bitstring, E: bitstring,
          srvok: bitstring));
  (* Step 5: Verify Srv_OK *)
  if srvok = PRF(KRID, (CONST_SRVOK, sid, Nr, Ts, E)) then
  (* Step 5: Compute MR1 *)
  let mr1 = PRF(KRID, (CONST_MR1, sid, Nr, Ts, E)) in
  (* Step 6: R -> S : MR1, sid *)
  out(ch, (mr1, sid));
  (* Step 8: S -> R : PROCEED, sid *)
  in(ch, (proceed_val: bitstring, =sid));
  (* Step 9: Verify PROCEED *)
  if proceed_val = PRF(KRID, (CONST_PROCEED, sid, Nr, Ts, E))
  then
  event readerAuthServer(sid, Nr, Ts, E);
  (* Step 10: R -> T : sid, Nr, Ts, E *)
  out(ch, (sid, Nr, Ts, E));
  (* Step 12: T -> R : sid, Nt, PID, EarlyAuth *)
  in(ch, (=sid, Nt: bitstring, pid: bitstring,
          ea: bitstring));
  (* Step 13: R -> S : sid, Nt, PID, EarlyAuth *)
  out(ch, (sid, Nt, pid, ea));
  (* Step 15: S -> R : MR2, sid *)
  in(ch, (mr2: bitstring, =sid));
  (* Step 16: R -> T : MR2 *)
  out(ch, mr2);
  (* Step 18: T -> R : Auth_T, sid *)
  in(ch, (autht: bitstring, =sid));
  (* Step 19: R -> S : Auth_T, sid *)
  out(ch, (autht, sid));
  (* Step 21: S -> R : ACK, sid *)
  in(ch, (ack: bitstring, =sid));
  (* Step 22: Verify ACK *)
  if ack = PRF(KRID, (CONST_ACK, sid, Nr, Nt, Ts)) then
  0.

(* ============================================================ *)
(* SERVER PROCESS                                               *)
(* ============================================================ *)
let processServer() =
  (* Step 2: R -> S : Nr *)
  in(ch, Nr: bitstring);
  (* Step 3: Generate session parameters *)
  new sid: bitstring;
  new Ts: bitstring;
  let E = Ts in (* simplified: epoch derived from Ts *)
  (* Step 3: Compute Srv_OK *)
  let srvok = PRF(KRID, (CONST_SRVOK, sid, Nr, Ts, E)) in
  event serverStartsReader(sid, Nr, Ts, E);
  (* Step 4: S -> R : sid, Ts, E, Srv_OK *)
  out(ch, (sid, Ts, E, srvok));
  (* Step 6: R -> S : MR1, sid *)
  in(ch, (mr1: bitstring, =sid));
  (* Step 7: Verify MR1 *)
  if mr1 = PRF(KRID, (CONST_MR1, sid, Nr, Ts, E)) then
```

```
  (* Step 7: Compute PROCEED *)
  let proceed_val = PRF(KRID, (CONST_PROCEED, sid, Nr, Ts, E))
  in
  (* Step 8: S -> R : PROCEED, sid *)
  out(ch, (proceed_val, sid));
  (* Step 13: R -> S : sid, Nt, PID, EarlyAuth *)
  in(ch, (=sid, Nt: bitstring, pid: bitstring,
          ea: bitstring));
  (* Step 14: Verify PID *)
  if pid = PIDGEN(KT, E) then
  (* Step 14: Verify EarlyAuth *)
  if ea = PRF(KT, (CONST_EA, sid, Nr, Ts, Nt)) then
  event serverAuthTag(sid, Nr, Ts, Nt);
  (* Step 14: Compute MR2 *)
  let mr2 = PRF(KT, (CONST_MR2, sid, Nr, Ts, Nt)) in
  (* Step 15: S -> R : MR2, sid *)
  out(ch, (mr2, sid));
  (* Step 19: R -> S : Auth_T, sid *)
  in(ch, (autht: bitstring, =sid));
  (* Step 20: Verify Auth_T *)
  if autht = PRF(KT, (CONST_AT, sid, Nr, Ts, Nt)) then
  event serverComplete(sid, Nr, Ts, Nt);
  (* Step 20: Compute ACK *)
  let ack = PRF(KRID, (CONST_ACK, sid, Nr, Nt, Ts)) in
  (* Step 21: S -> R : ACK, sid *)
  out(ch, (ack, sid));
  0.

(* ============================================================ *)
(* TAG PROCESS                                                  *)
(* ============================================================ *)
let processTag() =
  (* Step 10: R -> T : sid, Nr, Ts, E *)
  in(ch, (sid: bitstring, Nr: bitstring, Ts: bitstring,
          E: bitstring));
  (* Step 11: Generate Nt *)
  new Nt: bitstring;
  (* Step 11: Compute PID and EarlyAuth *)
  let pid = PIDGEN(KT, E) in
  let ea = PRF(KT, (CONST_EA, sid, Nr, Ts, Nt)) in
  event tagStarts(sid, Nr, Ts, Nt);
  (* Step 12: T -> R : sid, Nt, PID, EarlyAuth *)
  out(ch, (sid, Nt, pid, ea));
  (* Step 16: R -> T : MR2 *)
  in(ch, mr2: bitstring);
  (* Step 17: Verify MR2 *)
  if mr2 = PRF(KT, (CONST_MR2, sid, Nr, Ts, Nt)) then
  event tagAuthServer(sid, Nr, Ts, Nt);
  (* Step 17: Compute Auth_T *)
  let autht = PRF(KT, (CONST_AT, sid, Nr, Ts, Nt)) in
  (* Step 18: T -> R : Auth_T, sid *)
  out(ch, (autht, sid));
  0.

(* ============================================================ *)
(* Main process - unbounded sessions                            *)
(* ============================================================ *)
process
  ( (!processReader()) | (!processServer()) | (!processTag()) )
```

Table 7. ProVerif verification results (unbounded sessions).

Query	Property	Result
not attacker(KRID)	Secrecy of K_{RID}	true
not attacker(KT)	Secrecy of K_T	true
not attacker(TID)	Secrecy of TID	true
readerAuthServer ⇒ serverStartsReader	Reader authenticates Server	true
serverAuthTag ⇒ tagStarts	Server authenticates Tag	true
tagAuthServer ⇒ serverAuthTag	Tag authenticates Server	true
serverComplete ⇒ tagAuthServer	Full session agreement	true

All seven queries returned `true`, confirming that the protocol satisfies the following properties for an unbounded number of concurrent sessions:

- **Secrecy:** The reader–server key K_{RID}, the per-tag key K_T, and the tag identifier TID remain confidential and cannot be derived by the adversary, even across arbitrarily many protocol executions.
- **Reader–Server authentication:** Whenever a reader completes authentication with the server (event *readerAuthServer*), the server must have initiated the corresponding session (event *serverStartsReader*) with matching parameters $(session\_id, N_r, T_s, E)$.
- **Server–Tag authentication:** Whenever the server accepts a tag (event *serverAuthTag*), the tag must have participated (event *tagStarts*) with matching parameters $(session\_id, N_r, T_s, N_t)$.
- **Tag–Server authentication:** Whenever a tag accepts the server (event *tagAuthServer*), the server must have previously authenticated the tag (event *serverAuthTag*) in the same session.
- **Full session agreement:** Whenever the server completes a full session (event *serverComplete*), the tag must have authenticated the server (event *tagAuthServer*), confirming end-to-end three-party agreement.

These results, obtained through unbounded analysis, strengthen the bounded Scyther verification of Section 4 and provide strong formal evidence that the protocol achieves mutual authentication and key secrecy under the Dolev–Yao attacker model regardless of the number of concurrent sessions.

5. Performance Analysis of the Proposed Protocol

The presented protocol provides strong security guarantees operating on symmetric-key-only primitives (PRFs, KDF, and lightweight encodings), which fit the resource-constrained nature of many RFID deployments. In healthcare settings, schemes like SecMAP [24] support privacy and forward-secrecy properties through quadratic-residue-based public key operations (e.g., modular squaring on both reader and tag sides), at a greater cost than purely symmetric key constructions. In contrast, we confine all our expensive operations to the back-end server in our protocol, based on symmetric primitives for tags and readers; hence, it is better suited for very low-cost tags with a much larger population, as in logistics and supply chain systems [2,3].

A number of RFID authentication schemes in the literature employ temporary identifiers or hashed pseudonyms to protect tag identity. The protocol by Xu et al. [2] for smart logistics, for example, uses changing identifiers to reduce traceability. Our design follows a similar approach but introduces epoch-based pseudonym precomputation and epoch tables that support constant-time lookups on the server side. This allows constant-time tag identification, instead of requiring the back-end to search through a large database of keys for every authentication attempt.

5.1. Computational Overhead

To analyze the computational complexity of the proposed protocol, we consider all cryptographic operations performed by each respective entity (namely, tag, reader and server) upon a full execution of an authentication session. Table 8 gives an overview of the number of operations for each party.

Table 8. Computational operations per authentication session.

Entity	PRF Operations	Other Operations	Lookup
Tag	4	1 PRNG	—
Reader	4	1 PRNG, 1 timestamp check	—
Server	7	1 KDF, 1 PRNG	$O(1)$ hash

Note: Cost tier (typical): bitwise $\ll$ PRNG $\ll$ PRF. Counts are primitive calls.

5.1.1. Tag Operations

Since the tag has the least amount of computing power available, it is probably doing the least amount of work. Essentially, the tag then, in one authentication session, does:

- One left rotate operation to compute $HPID$. This is an ultralight operation whose execution time is considered negligible.
- Two PRF evaluations to compute the epoch pseudonym PID and early authentication token $EarlyAuth$ (Step 11).
- One PRF verification (Step 17) for server authentication with respect to MR_2.
- One PRF evaluation for final authentication token $Auth_T$ (Step 17).

Overall, the tag executes four PRF operations for each session. It is worth mentioning that the tag does not involve key derivation, hash table look-up or any other complex encoding operation except for the injective encoding function $\mathsf{encode}(\cdot)$ (which can be a simple concatenation with length prefixes). The tag also produces one random nonce N_t, with a call to a pseudorandom number generator (PRNG).

5.1.2. Reader Operations

A reader mainly acts as a relay and runs proof-of-authentication checks for tokens received from the server. In one particular session, the reader runs:

- One PRNG call to generate the fresh nonce N_r (Step 1);
- Three PRF verifications to authenticate the server: checking Srv_OK (Step 5), verifying $PROCEED$ (Step 9), and validating the final ACK (Step 22);
- One PRF computation for generating the reader proof MR_1 (Step 5);
- One timestamp validation comparing $|T_{\text{current}} - T_s| \leq \Delta T$ (Step 5).

Thus, the reader needs to do four PRF operations (one computation and three verifications), **one call to PRNG**, and **one timestamp check** for each session. The reader does not have any cryptographic state other than the key K_{RID} that is shared with the server and does not perform database lookups.

5.1.3. Server Operations

Although most of the calculations are done on the server, it is not a real issue since the back-end is expected to have high processing capabilities. In one session, the server runs:

- One right rotate operation to compute PID. This is an ultralight operation whose execution time is considered negligible.
- One KDF operation used to derive the per-tag key K_T from K_{master} and the identifier for the tag (TID, which can be cached).
- Up to three hash table lookups in the epoch maps $MAP[E]$, $MAP[E-1]$ and $MAP[E-2]$ for constant-time tag identification (Step 14); in the common case (no clock drift), the first lookup in $MAP[E]$ succeeds.
- Seven PRF computations for creating and checking the authentication tokens: Srv_OK (Step 3), verification of MR_1 (Step 7), $PROCEED$ (Step 7), verification of $EarlyAuth$ (Step 14), MR_2 (Step 14), verification of $Auth\_T$ (Step 20), and sending back an acknowledgment, as in (ACK, Step 20).

- One random number generation for *session_id* (Step 3).

The session incurs a cost of seven PRF operations, 1 KDF operation and up to three hash table lookups on the server. The hash table lookups are handled in constant time $O(1)$, which is a significant optimization versus linear search protocols that would require $O(N)$ operations for N tags.

5.1.4. Precomputation Phase

We assume the server runs an offline precomputation phase to initialize epoch tables before each epoch E. For each of the N registered tags, the server computes:

- One KDF operation to derive K_T from K_{master} and TID.
- One PRF operation to compute $PID = \text{Trunc}_{96}(PRF_{K_T}(\text{“TIDL”} \parallel E))$.

This gives us N KDF computations and N PRF computations per epoch. This preprocessing, however, is performed offline once before each epoch starts (e.g., every 10 min), and the KDF results could be cached over epochs since K_T does not change. Hence, the cost per authentication session, amortized over all the sessions, becomes negligible.

5.1.5. Computational Comparison with Related Work

In comparison to heavyweight authentication protocols based on elliptic curve cryptography (ECC) like the protocol by Sharma et al. [27], the proposed protocol requires vastly less computational power on RFID tags with limited resources. ECC-based schemes usually rely on point multiplication, which is computationally intensive and might produce high latency on cheap tags. In particular, lightweight symmetric key protocols such as those by Wang et al. [28] and Xu et al. [2] replace costly public key operations with hash- or PRF-based mechanisms and pseudorandom number generation, making tag-side computational cost lower. But such schemes typically require server-side exhaustive search to locate the tags, incurring linear-time lookup complexity and limited scalability in large deployments.

Zhu et al. [14] also rely on symmetric key primitives but employ hash-based indexing to achieve constant-time $O(1)$ lookup, similar to our approach (but with different cryptographic primitives).

Ultra-lightweight protocols like that used by Alhasan et al. [17] reduce computation even further by using only simple bitwise operations (e.g., XOR, rotation, AND, OR). These designs are highly efficient, but generally fall back to linear server-side lookup and offer limited security features compared with PRF-based schemes.

In contrast, the protocol we propose strikes an excellent balance between security and efficiency. While it carries out reasonable PRF computations relative to other symmetric key designs, its constant-time server-side lookup through pseudonym indexing by epochs makes it practical for large RFID systems with extensive decoding and storage potential.

Indeed, as highlighted in Table 9, although the performance of the proposed protocol is competitive with respect to the number of PRF operations among other symmetric key schemes, it considerably benefits from epoch-based pseudonym indexing, which yields $O(1)$ constant-time server lookup independent of the total number of registered tags N. This feature enables the protocol to efficiently scale for large deployments with millions of tags (e.g., in large-scale logistics or healthcare systems).

Among the $O(1)$ schemes in Table 9, heavyweight approaches, such as that used by Sharma et al. [27], achieve constant-time lookup by recovering a static per-tag value during each session, while our protocol achieves $O(1)$ via rotating epoch pseudonyms, preventing cross-epoch linkability even by parties holding server keys.

Table 9. Computational overhead comparison with related work.

Protocol	Tag Ops	Reader Ops	Server Ops	Lookup
SecMAP [24]	2H + 1MS + 1P	2H + 1P + 1MS	4H + 1P + 2SR	$O(1)$
Sharma et al. [27]	6SM + 4PA + 2H	N/A †	6SM + 2PA + 1PS + 2H	$O(1)$
Wang et al. [28]	2 PRNG	4 PRNG	2 PRNG	$O(N)$
Xu et al. [2]	2 Eac + 2 Rot	Random + 5 Hash	Random + 5 Hash	$O(N)$
Zhu et al. [14]	5 PRNG	5 PRNG	9 PRNG	$O(1)$
Alhasan et al. [17]	9 XOR + 2 Rot + 2 AND + 1 OR	2 XOR + 1 AND	6 XOR + 2 Rot + 1 AND + 1 OR	$O(N)$
Proposed	**4 PRF**	**4 PRF**	**7 PRF + 1 KDF + 1 P**	$\boldsymbol{O(1)}$

Bitwise operations: Xu et al. [2] use Rot and Eac; Alhasan et al. [17] use XOR, AND, OR, Rot. Bitwise costs treated as negligible in time-dominant comparisons. H: Hash; MS: modular squaring; P: PRNG; SR: square root solving; SM: elliptic curve scalar multiplication; PA: point addition; PS: point subtraction; PRF: pseudorandom function. † Sharma et al. [27] do not report reader-side computational cost.

5.1.6. Execution Time Benchmarks

To complement the operation count comparison in Table 9, we estimate concrete per-session server-side execution times using the timing measurements reported by Zhou et al. [23]: hash operation $H = 0.253$ ms, pseudorandom number generation $P = 0.021$ ms, modular squaring $MS = 1.896$ ms, and square root solving $SR = 3.481$ ms. Since a PRF can be instantiated as HMAC (two hash invocations), we estimate $PRF \approx 2H = 0.506$ ms. The KDF is estimated as one hash invocation (0.253 ms). Bitwise operations (XOR, rotation, AND, OR) are treated as individually negligible.

For protocols with $O(N)$ server-side search, the total server time includes the base cryptographic computation plus the average linear search cost of $N/2$ verification operations per candidate tag. Table 10 presents the estimated server-side execution times at different deployment scales.

Table 10. Server-side execution time comparison across deployment scales (in milliseconds).

Protocol	Lookup	Server base	$N = 10^5$	$N = 5 \times 10^5$	$N = 10^6$
SecMAP [24]	$O(1)$	7.995	7.995	7.995	7.995
Sharma [27]	$O(1)$	>20.0	>20.0	>20.0	>20.0
Wang [28]	$O(N)$	0.042	1050	5250	10,500
Xu [2]	$O(N)$	1.286	12,651	63,251	126,501
Zhu [14]	$O(1)$	0.189	0.189	0.189	0.189
Alhasan [17]	$O(N)$	<0.01	∼50	∼250	∼500
Proposed	$\boldsymbol{O(1)}$	**3.816**	**3.816**	**3.816**	**3.816**

Notes: Timing references from Zhou et al. [23]: $H = 0.253$ ms, $P = 0.021$ ms, $MS = 1.896$ ms, $SR = 3.481$ ms. $PRF \approx 2H = 0.506$ ms; $KDF \approx H = 0.253$ ms. Server base is the per-session cryptographic cost excluding search: SecMAP: $4H + 1P + 2SR = 7.995$; Wang: $2P = 0.042$; Xu: $1P + 5H = 1.286$; Zhu: $9P = 0.189$; Proposed: $7 \times PRF + 1 \times KDF + 1P = 3.816$. For $O(N)$ protocols, the N columns include an average linear search of $N/2$ candidates, each requiring one verification operation: $1P = 0.021$ ms for Wang, $1H = 0.253$ ms for Xu, and ∼0.001 ms (bitwise + memory access) for Alhasan. For reference, Wang at $N = 10^6$: 10,500 ms = 10.5 s; Xu at $N = 10^6$: 126,501 ms ≈ 2.1 min. Sharma's reader cost is not reported in their paper; ECC timing estimated conservatively.

The results demonstrate the critical impact of lookup complexity on server-side scalability. At $N = 10^6$ tags, the $O(N)$ protocols require between 0.5 and 126.5 s per authentication session on the server, while the proposed protocol completes in a fixed 3.816 ms regardless of the number of registered tags. Even among $O(1)$ protocols, the proposed protocol's server cost (3.816 ms) is lower than SecMAP (7.995 ms) and substantially lower than Sharma (>20 ms), while Zhu (0.189 ms) achieves a faster base time but lacks three-party authentication and epoch-based unlinkability (see Table 1).

Including tag and reader costs, the proposed protocol's total per-session time is 7.906 ms (referring to Tables 9 and 10, this is equivalent to 4 PRF + 4 PRF + 7 PRF +

1 KDF + 1 P). This is well below the human-perceptible latency threshold of approximately 200 ms—and remains fixed regardless of deployment scale.

5.2. Communication Overhead

Communication overhead is a critical metric for RFID systems because passive tags have limited power budgets and can only respond when energized by the reader's electromagnetic field. Minimizing the number and size of messages extends the operational range and reliability of the system.

5.2.1. Message Count and Flow

The proposed protocol requires 12 message transmissions in a complete authentication session, as detailed in Tables 3 and 4:

1. Reader → Server: $\{RID, N_r\}$;
2. Server → Reader: $\{session\_id, T_s, E, Srv\_OK\}$;
3. Reader → Server: $\{MR_1, session\_id\}$;
4. Server → Reader: $\{PROCEED, session\_id\}$;
5. Reader → Tag: $\{session\_id, N_r, T_s, E\}$;
6. Tag → Reader: $\{session\_id, N_t, HPID, EarlyAuth\}$;
7. Reader → Server: $\{session\_id, N_t, HPID, EarlyAuth\}$;
8. Server → Reader: $\{MR_2, session\_id\}$;
9. Reader → Tag: $\{MR_2\}$;
10. Tag → Reader: $\{Auth\_T, session\_id\}$;
11. Reader → Server: $\{Auth\_T, session\_id\}$;
12. Server → Reader: $\{ACK, session\_id\}$.

Of these 12 messages, four messages involve the tag (messages 5, 6, 9, 10), which constitute the critical bottleneck. The remaining eight messages occur over the more capable reader–server channel (typically high-speed wireless).

5.2.2. Data Size Analysis

Let us estimate the byte sizes for typical implementations:

- RID: 32 bits (4 bytes)—reader identifier;
- TID: 96 bits (12 bytes)—tag identifier (EPC Gen2 standard);
- N_r, N_t: 64 bits each (8 bytes)—random nonces;
- T_s: 32 bits (4 bytes)—Unix timestamp;
- E: 32 bits (4 bytes)—epoch index (ensures unique epochs for the full Unix timestamp range);
- $session\_id$: 64 bits (8 bytes)—unique session identifier;
- $HPID$: 96 bits (12 bytes);
- **PRF outputs**: 128 bits each (16 bytes)—standard output length for lightweight symmetric key authentication tokens.

Reader–Tag Communication (Critical Path)

- **Message 5** (Reader → Tag): $session\_id$ (8) + N_r (8) + T_s (4) + E (4) = **24 bytes**;
- **Message 6** (Tag → Reader): $session\_id$ (8) + N_t (8) + $HPID$ (12) + $EarlyAuth$ (16) = **44 bytes**;
- **Message 9** (Reader → Tag): MR_2 (16) = **16 bytes**;
- **Message 10** (Tag → Reader): $Auth\_T$ (16) + $session\_id$ (8) = **24 bytes**.

Total Reader–Tag communication: 24 + 44 + 16 + 24 = **108 bytes** (bidirectional)

This is well within the capabilities of extended EPC Class-1 Generation-2 (EPC C1G2) RFID tags that incorporate lightweight symmetric key coprocessors, which typically sup-

port messages up to 512 bits (64 bytes) in a single transmission. The tag sends two messages (44 bytes and 24 bytes, respectively), each well within the single-message limit.

Reader–Server Communication

- **Message 1**: RID (4) + N_r (8) = **12 bytes**;
- **Message 2**: $session\_id$ (8) + T_s (4) + E (4) + Srv_OK (16) = **32 bytes**;
- **Message 3**: MR_1 (16) + $session\_id$ (8) = **24 bytes**;
- **Message 4**: $PROCEED$ (16) + $session\_id$ (8) = **24 bytes**;
- **Message 7**: $session\_id$ (8) + N_t (8) + $HPID$ (12) + $EarlyAuth$ (16) = **44 bytes**;
- **Message 8**: MR_2 (16) + $session\_id$ (8) = **24 bytes**;
- **Message 11**: $Auth\_T$ (16) + $session\_id$ (8) = **24 bytes**;
- **Message 12**: ACK (16) + $session\_id$ (8) = **24 bytes**.

Total Reader–Server communication: 12 + 32 + 24 + 24 + 44 + 24 + 24 + 24 = **208 bytes** (bidirectional)

Note that some data is relayed by the reader between the tag and the server. The total data transmitted across all channels is **316 bytes** (208 + 108 = 316). Three relay overlaps exist: 44 bytes (messages 6→7, tag response forwarded to server), 16 bytes (messages 8→9, MR_2 forwarded to tag), and 24 bytes (messages 10→11, $Auth\_T$ forwarded to server), totaling 84 bytes of duplicated data. The unique data generated is therefore approximately **232 bytes**.

5.2.3. Communication Comparison with Related Work

Table 11 compares the communication overhead of the proposed protocol with existing schemes. The proposed protocol has higher communication overhead compared to some lightweight schemes due to the explicit three-party mutual authentication and the additional server–reader handshake. However, this increased overhead provides significant security benefits:

1. **Explicit mutual authentication** among all three parties (server, reader, tag);
2. **Session confirmation** through the final token ACK;
3. **Reader authorization** via the $PROCEED$ token;
4. **Constant-time tag identification** using epoch pseudonyms.

Table 11. Communication overhead comparison with related work.

Protocol	Interaction Times	Tag Data (Bytes)	Total Data (Bytes)
SecMAP [24]	7	∼164	∼456
Sharma et al. [27]	3	80	160
Wang et al. [28]	7	∼60	∼108
Xu et al. [2]	5	144	405
Zhu et al. [14]	7	∼60	∼168
Fan et al. [12]	9	∼36	∼284
Alhasan et al. [17]	5	∼72	∼132
Proposed	**12**	**108**	**316**

Xu et al. [2] define communication cost as interaction times plus communication data length (see Table 5 in their study). For Xu et al., 144 bytes corresponds to $l_k + l_{id} = 1024 + 128 = 1152$ bits (see Tables 4 and 7 in their paper). Alhasan et al. [17] report $6L$ bits for the wireless tag–reader channel (see Section IV-B of their paper); with $L = 96$ bits (see Section II-B of their paper), this is 72 bytes (Tag Data column). Including the reader–server channel adds $3L$ (n_1, M_3, M_4 in their Step 3) and $2L$ (M_5, M_6 in their Step 4), yielding $11L = 132$ bytes total (Total Data column). Sharma et al. report 640 bits each for tag and server communication (see Table 6 in their paper).

For scenarios where security and scalability come first (healthcare, high-value logistics), the additional communication overhead represents a justified trade-off. Also, reader–server

communication occurs over high-bit-width channels rather than over a bottleneck. For RFID tags with lightweight symmetric key support, the tag communication (108 bytes total) is practical.

5.3. Storage Overhead

Storage requirements vary significantly across the three entities in the RFID system, with tags having the most stringent constraints.

5.3.1. Tag Storage

Each tag must store the following data in non-volatile memory:

1. **Tag identifier** (TID): 96 bits (12 bytes)—permanent unique identifier;
2. **Derived key** (K_T): 128 bits (16 bytes)—derived from K_{master} and TID;
3. **Nonce** (N_{ST}): 64 bits (8 bytes)—number of rotation bits used in generating $HPID$;
4. **Domain separation constants**: Three constants used in tag operations:
   - CONST_EA: 128 bits (16 bytes);
   - CONST_MR2: 128 bits (16 bytes);
   - CONST_AT: 128 bits (16 bytes).
5. **Pseudonym label** ("TIDL"): 32 bits (4 bytes)—fixed ASCII string used in PID derivation.

Note: Storage overhead refers to the additional memory required for authentication data (keys, identifiers, constants) and server-side indexing structures (epoch maps), excluding application-specific data.

Total tag storage: 12 + 16 + 8 + 48 + 4 = **88 bytes** The stateless tag design—storing only TID, K_T, three domain separation constants, and the pseudonym label—achieves a low memory footprint and avoids session state overhead, which is particularly beneficial for low-resource devices. Importantly, the tag does not need to store:

- Session state (nonces, session IDs) beyond the current session;
- Multiple keys or key versions;
- Any history or counters that could desynchronize.

This stateless design ensures robustness against power loss and desynchronization attacks.

5.3.2. Reader Storage

Each reader must store:

1. **Reader identifier** (RID): 32 bits (4 bytes);
2. **Shared key with server** (K_{RID}): 128 bits (16 bytes);
3. **Domain separation constants**: Four constants used in reader–server authentication:
   - CONST_SRV_OK: 128 bits (16 bytes);
   - CONST_MR1: 128 bits (16 bytes);
   - CONST_PROCEED: 128 bits (16 bytes);
   - CONST_ACK: 128 bits (16 bytes).

Total reader storage: 4 + 16 + 64 = **84 bytes**

In any case, readers usually have megabytes or gigabytes of memory, so this overhead is small. Session state: the reader may also cache session state during active sessions, but that is only temporary storage and not security-critical.

5.3.3. Server Storage

The storage requirements are highest on the server, and scale according to the number of tags and readers in the system. For a deployment with N tags and M readers, the server has to maintain:

1. **Master secret** (K_{master}): 128 bits (16 bytes)—single global secret;
2. **Reader keys**: $M \times 128$ bits—one key K_{RID} per reader;
3. **Tag identifiers and derived keys**: $N \times (96 + 128)$ bits = $N \times 224$ bits:
   - Each tag requires storage of TID (96 bits) and K_T (128 bits);
   - Alternatively, K_T can be derived on-demand from K_{master} and TID, requiring only $N \times 96$ bits.
4. **Epoch maps**: $3 \times N$ hash table entries, as shown in Table 5:
   - Three tables ($MAP[E]$, $MAP[E-1]$, and $MAP[E-2]$) for clock drift tolerance;
   - Each table is indexed by $hash(PID)$; each entry stores only the corresponding TID (the PID is not retained, as collision resolution is performed via EarlyAuth verification);
   - Hash table overhead depends on implementation (load factor, collision handling).
5. **Domain separation constants**: Seven constants (total 896 bits = 112 bytes)
6. **Nonce** (N_{ST}): 64 bits (8 bytes)—number of rotation bits used in regenerating PID.

As an example, for a large deployment with:

- $N =$ 1,000,000 tags (1 million);
- $M =$ 100 readers.

Therefore, on the server, we have the following storage: **Reader key storage**:

- 100 × 16 bytes = **1.6 KB**.

Tag data storage: (storing both TID and K_T):

- 1,000,000 × 28 bytes = **28 MB**.

Epoch map storage:

- Each hash table entry stores TID (12 bytes) plus overhead, which is needed to manage colliding entries: linked-list pointers and memory alignment. The $hash(PID)$ index is built at lookup time and never actually stored; the physical PID is suppressed, since verification of $EarlyAuth$ shows us which tag we need.
- Per-entry breakdown: TID = 12 B, next-pointer = 8 B (64-bit), memory-alignment padding = 4 B, bucket metadata = 4 B; total = $12 + 8 + 4 + 4 = 28$ bytes.
- Combined result at 28 bytes per entry for each epoch map: 3 × 1,000,000 × 28 bytes = **84 MB**.

Finally, the total server storage is: 16 B + 1.6 KB + 28 MB + 84 MB + 112 B + 8 B ≈ **112 MB** for 1 million tags (reader key contribution is negligible).

Table 12 provides a quantitative evaluation of server storage requirements across multiple deployment scales.

The storage overhead grows linearly with N and remains practical for modern server infrastructure. The three-epoch storage accounts for approximately 75% of the total server memory (e.g., 80 MB out of 107 MB for one million tags), which is a modest cost for the benefit of constant-time $O(1)$ lookup and 30 min clock drift tolerance.

This is remarkably efficient compared to protocols that require $O(N)$ online searches. For comparison:

- A protocol that stores only TID and K_T but requires a linear search would need ∼28 MB of data but would perform $O(N)$ computations per session.
- Our protocol trades ∼84 MB additional storage for constant-time $O(1)$ lookup, a highly favorable trade-off for modern servers with gigabytes of RAM.

Table 12. Server storage requirements across deployment sizes ($M = 100$ readers).

Tags (N)	Tag Data (MB)	Epoch Maps (MB)	Total (MB)
1000	0.03	0.08	0.11
10,000	0.28	0.84	1.12
100,000	2.80	8.40	11.20
500,000	14.00	42.00	56.00
1,000,000	28.00	84.00	112.00

Notes: Tag Data = $N \times 28$ bytes ($TID + K_T$). Epoch Maps = $3 \times N \times 28$ bytes (three hash tables $MAP[E]$, $MAP[E-1]$, $MAP[E-2]$, at ∼28 bytes per entry including overhead). Even at 1 million tags, the total (∼107 MB) represents only 1.3% of 8 GB RAM.

5.3.4. Storage Comparison with Related Work

Table 13 compares storage requirements across different protocols. Our protocol's server per-tag storage (112 bytes) is comparable to or lower than that of other constant-time lookup protocols, such as SecMAP (128 bytes), and even slightly lower than several linear search protocols, including those by Alhasan et al. [17] (120 bytes) and Fan et al. [12] (280 bytes), while still providing constant-time $O(1)$ tag identification. The modest server overhead (e.g., 112 MB for one million tags) easily fits modern infrastructure, and the trade-off in favor of $O(1)$ lookup becomes increasingly valuable as the tag population grows. The tag storage needed (88 bytes) is modest and is well within the capacity of contemporary RFID tags.

Table 13. Storage overhead comparison with related work.

Protocol	Tag (Bytes)	Server Per-Tag (Bytes)	Lookup
SecMAP [24]	∼256	∼128	$O(1)$
Sharma et al. [27]	200 †	40 ‡	$O(1)$
Wang et al. [28]	∼12	∼24	Linear
Xu et al. [2]	144	—	Linear
Zhu et al. [14]	∼24	∼48	$O(1)$
Fan et al. [12]	∼140	∼280	Linear
Alhasan et al. [17]	∼84 *	∼120 *	Linear
Proposed	**88**	**112** §	$\boldsymbol{O(1)}$

* Alhasan tag storage: $7L$ bits as reported in Section IV-A of their paper; $L = 96$ yields 84 bytes. Server per-tag storage is not reported in their paper, but per Section II-B(4), the server keeps old and new entries of K_1, K_2, and IDS, plus ID, SID, T_{S1}, T_t, totaling $10L = 120$ bytes per tag (one-time database storage; no indexing structure since lookup is linear). † Sharma: 1600 bits tag storage. ‡ Sharma: 40 bytes/tag + 180 bytes fixed. § Our protocol: 112 bytes/tag (= 28 bytes tag data + 84 bytes epoch maps) + ∼136 bytes fixed (K_{master} + 7 domain separation constants + N_{ST}) + 16 bytes per reader. Note: Storage overhead: the extra memory needed for authentication and server-side indexing data structures; excludes application-specific data.

5.4. Experimental Scalability Validation

To validate the claimed $O(1)$ server-side lookup performance, we implemented the epoch-based pseudonym indexing mechanism and measured tag identification time across deployments ranging from 10^3 to 10^6 tags. The PRF was instantiated with HMAC-SHA-256 truncated to 96 bits, and the epoch hash table was implemented using a standard hash map with separate chaining for collision resolution. For each deployment size N, we generated N tags with unique TIDs and per-tag keys $K_T = KDF(K_{\text{master}}, TID)$, built the epoch hash map $MAP[E]$, and measured the average lookup time over 1000 independent tag identification operations. Each lookup consists of computing $hash(PID)$ from the received pseudonym, retrieving the candidate TID from the bucket, and verifying the match via *EarlyAuth*. The results were compared against a baseline linear search that iterates over all N tag records and performs a PRF verification for each.

The results in Table 14 confirm the $O(1)$ property of the epoch-based lookup. From $N = 1000$ to $N = 100{,}000$, the hash table lookup time remains effectively constant at approximately 10 µs. At $N = 1{,}000{,}000$, the lookup time increases to 37 µs due to CPU cache effects when the epoch table exceeds the processor's L3 cache capacity; however, this remains an algorithmically constant-time operation (the same number of hash and PRF computations are performed regardless of N), and the speedup over linear search still exceeds 135,000×. In contrast, the linear search time scales linearly with N, reaching over 5 s for one million tags. No hash collisions were observed at any deployment scale, consistent with the theoretical birthday collision probability of $N^2/2^{97} \approx 6.3 \times 10^{-18}$ for $N = 10^6$.

Table 14. Lookup time: epoch-based $O(1)$ vs. linear $O(N)$ search.

Tags (N)	Hash Lookup (µs): $O(1)$	Linear Search (µs): $O(N)$	Speedup
1000	10.67	2061	193×
10,000	10.67	21,948	2058×
100,000	10.16	287,173	28,260×
1,000,000	37.26	5,048,186	135,487×

6. Conclusions

In this paper, we presented a novel lightweight RFID authentication scheme capable of providing the following significant contributions:

1. **True three-party mutual authentication** with explicit cryptographic binding among all three pairs—server to reader, reader to tag, and server to tag—the server verifies that the tag is genuine using its own processes rather than trusting the assertions made by the intermediate reader.
2. **Constant-time server scalability:** Epoch-based pseudonym indexing achieves constant time complexity for tag identification $O(1)$ regardless of how many millions of tags the same entity has registered, as opposed to linear search $O(N)$ in many existing lightweight protocols.
3. **Strong security guarantees** in the Dolev–Yao attacker model, resistant to replay, impersonation, man-in-the-middle and desynchronization attacks, formally verified using Scyther and validated through unbounded ProVerif analysis.
4. **Epoch-based unlinkability** preventing cross-epoch tracking while remaining within the computational budget of RFID tags equipped with lightweight symmetric key coprocessors.
5. **Practical deployability** with only four PRF operations on tags, 88 bytes of tag storage, and compatibility with resource-constrained RFID tags supporting lightweight symmetric key operations.

The performance evaluation shows that the protocol achieves a good trade-off between security and efficiency. Although the communication cost (316 bytes in total, 108 bytes for tag–reader) is larger than some ultra-lightweight protocols like those used by Wang et al. [28] (108 bytes) and Alhasan et al. [17] (132 bytes), it remains lower than that of SecMAP [24] (456 bytes) and that used by Xu et al. [2] (405 bytes). This comes at the cost of stronger security properties and much higher scalability.

The server storage overhead (∼112 MB per million tags) is low for modern infrastructure, and it supports constant-time authentication, which is a crucial feature in large-scale deployments.

The protocol is ideal for applications with high authentication strength and moderate privacy needs in resource-constrained settings, such as healthcare (e.g., patient tracking,

medication management), logistics (i.e., supply chain monitoring), access control, and IoT sensor networks. The three-epoch window provides clock drift tolerance of up to ∼30 min, enabling operation in hardware deployments where perfect clock synchronization cannot be guaranteed.

For future work, additional formal analysis using tools such as Tamarin or BAN logic could be conducted to further strengthen security assurances. Moreover, it would be interesting to explore protocol extensions for particular application domains, e.g., group authentication for batch tag reading, anonymous authentication for privacy-critical scenarios, or integration with blockchain or distributed ledger technologies, to enable enhanced auditability in supply chain applications.

Author Contributions: P.E.A.-C.: conceptualization, methodology, validation, writing—review and editing, supervision; M.A.H.: methodology, validation, writing—original draft; M.A.-H.: methodology, validation, writing—review and editing, coordination. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Islam, S.M.R.; Kwak, D.; Kabir, M.H.; Hossain, M.; Kwak, K.S. The Internet of Things for health care: A comprehensive survey. *IEEE Access* **2015**, *3*, 678–708. [CrossRef]
2. Xu, C.; Wei, W.; Zheng, S. Efficient mobile RFID authentication protocol for smart logistics targets tracking. *IEEE Access* **2023**, *11*, 4322–4336. [CrossRef]
3. Anandhi, S.; Anitha, R.; Sureshkumar, V. IoT enabled RFID authentication and secure object tracking system for smart logistics. *Wirel. Pers. Commun.* **2018**, *104*, 543–560. [CrossRef]
4. Finkenzeller, K. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication*, 3rd ed.; Wiley: Hoboken, NJ, USA, 2010.
5. Sun, Z.; Ren, Z.; Yan, H. Modern tracking technology of logistics information research progress review. *J. Zhejiang Univ. Sci. Technol.* **2005**, *17*, 126–130.
6. Wang, W.C.; Yona, Y.; Diggavi, S.N.; Gupta, P. Design and analysis of stability-guaranteed PUFs. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 978–992. [CrossRef]
7. Liu, D.; Ling, J.; Yang, X. An improved RFID authentication protocol to meet the backward privacy. *Comput. Sci.* **2016**, *43*, 128–130.
8. Juels, A. RFID security and privacy: A research survey. *IEEE J. Sel. Areas Commun.* **2006**, *24*, 381–394. [CrossRef]
9. Avoine, G.; Dysli, E.; Oechslin, P. Reducing time complexity in RFID systems. In Proceedings of the 12th International Conference on Selected Areas in Cryptography (SAC), Kingston, ON, Canada, 11–12 August 2005; pp. 291–306.
10. Baashirah, R.; Abuzneid, A. Survey on prominent RFID authentication protocols for passive tags. *Sensors* **2018**, *18*, 3584. [CrossRef] [PubMed]
11. Fan, K.; Jiang, W.; Li, H.; Yang, Y. Lightweight RFID protocol for medical privacy protection in IoT. *IEEE Trans. Ind. Inform.* **2018**, *14*, 1656–1665. [CrossRef]
12. Fan, K.; Zhu, S.; Zhang, K.; Li, H.; Yang, Y. A lightweight authentication scheme for cloud-based RFID healthcare systems. *IEEE Netw.* **2019**, *33*, 44–49. [CrossRef]
13. Shimizu, K.; Wang, S.; Kai, H.; Takahashi, H.; Shimizu, A. A lightweight and secure one-time RFID authentication protocol based on SAS-L2. In Proceedings of the 2024 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), Danang, Vietnam, 3–6 November 2024; pp. 1–4.
14. Zhu, F.; Li, P.; Xu, H.; Wang, R. A novel lightweight authentication scheme for RFID-based healthcare systems. *Sensors* **2020**, *20*, 4846. [CrossRef] [PubMed]
15. Ma, Z.; Cheng, L. Mobile mutual authentication protocol based on word synthesis operation. *Appl. Res. Comput.* **2017**, *33*, 814–819.
16. Huang, K.; Liu, Y.; Yin, X. Ultra-lightweight RFID mutual authentication protocol based on regeneration transformation. *J. Comput. Appl.* **2019**, *39*, 118–125.

17. Alhasan, A.Q.A.; Rohani, M.F.; Abu-Ali, M.S. Ultra-lightweight mutual authentication protocol to prevent replay attacks for low-cost RFID tags. *IEEE Access* **2024**, *12*, 50925–50934. [CrossRef]
18. Akiirne, Z.; Sghir, A.; Bouzidi, D. UDAP: Ultra-lightweight dot product-based authentication protocol for RFID systems. *Cybersecurity* **2024**, *7*, 68. [CrossRef]
19. Tan, X.; Dong, M.; Wu, C.; Ota, K.; Wang, J.; Engels, D.W. An energy-efficient ECC processor of UHF RFID tag for banknote anti-counterfeiting. *IEEE Access* **2017**, *5*, 3044–3054. [CrossRef]
20. Liu, B.; Yang, B.; Su, X. An improved two-way security authentication protocol for RFID system. *Information* **2018**, *9*, 86. [CrossRef]
21. Zhu, H.; Zhang, Y. An improved RFID authentication protocol with privacy protection based on chaotic maps. *J. Internet Technol.* **2018**, *19*, 1777–1787.
22. Akgun, M.; Bayrak, A.O.; Caglayan, M.U. Attacks and improvements to chaotic map-based RFID authentication protocol. *Secur. Commun. Netw.* **2015**, *8*, 4028–4040.
23. Zhou, Z.; Wang, P.; Li, Z. A quadratic residue-based RFID authentication protocol with enhanced security for TMIS. *J. Ambient. Intell. Humaniz. Comput.* **2019**, *10*, 3603–3615.
24. Zhu, F. SecMAP: A secure RFID mutual authentication protocol for healthcare systems. *IEEE Access* **2020**, *8*, 192192–192205. [CrossRef]
25. Xu, H.; Yin, X.; Zhu, F.; Li, P. An enhanced secure authentication scheme with one more tag for RFID systems. *IEEE Sens. J.* **2021**, *21*, 17189–17199. [CrossRef]
26. Yang, A.; Boshoff, D.; Hu, Q.; Hancke, G.P.; Luo, X.; Weng, J.; Mayes, K.; Markantonakis, K. Privacy-preserving group authentication for RFID tags using bit-collision patterns. *IEEE Internet Things J.* **2021**, *8*, 11607–11620. [CrossRef]
27. Sharma, S.; Kaushik, B.; Rahmani, M.K.I.; Ahmed, M.E. Cryptographic solution-based secure elliptic curve cryptography enabled radio frequency identification mutual authentication protocol for Internet of Vehicles. *IEEE Access* **2021**, *9*, 147114–147128. [CrossRef]
28. Wang, Y.; Shu, F.; Lei, X.; Wang, X.; Dong, R.; Cheng, Q. A lightweight RFID security authentication protocol for smart medical systems. In Proceedings of the 2023 Eleventh International Conference on Advanced Cloud and Big Data (CBD), Danzhou, China, 18–19 December 2023; pp. 69–74.
29. Li, T.; Weng, C.Y.; Lee, C.C. A secure RFID tag authentication protocol with privacy preserving in telecare medicine information system. *J. Med. Syst.* **2015**, *39*, 77. [CrossRef] [PubMed]
30. Xiao, F. A provable secure RFID mutual authentication protocol. In Proceedings of the 2024 IEEE 4th International Conference on Electronic Technology, Communication and Information (ICETCI), Changchun, China, 24–26 May 2024; pp. 62–66.
31. Cremers, C.J.F. The Scyther Tool: Verification, falsification, and analysis of security protocols. In Proceedings of the 20th International Conference on Computer Aided Verification (CAV), Princeton, NJ, USA, 7–14 July 2008; pp. 414–418.
32. Blanchet, B. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*; IEEE: New York, NY, USA, 2001; pp. 82–96.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.