A Flyweight RFID Authentication Protocol

MIKE BURMESTER
Florida State University, Tallahassee
and
JORGE MUNILLA[1]
Universidad de Málaga, Spain

We propose a lightweight RFID authentication protocol that supports forward and backward security. The only cryptographic mechanism that this protocol uses is a pseudo-random number generator (PRNG) that is shared with the backend Server. Authentication is achieved by exchanging a few numbers (3 or 5) drawn from the PRNG. The protocol is optimistic with constant lookup time, and can be easily adapted to prevent online man-in-the-middle relay attacks. Security is proven in the UC security framework.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*; K.6.5 [**Management of Computing and Information Systems**]: Miscellaneous—*Security*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Design, Security, Theory

Additional Key Words and Phrases: RFID, EPCGen2, authentication, forward security, backward security, universal composability

1. INTRODUCTION

Radio Frequency Identification (RFID) is a promising new technology that is widely deployed for supply-chain and inventory management, retail operations and more generally, automatic identification. The advantage of RFID over barcode technology is that it does not require direct line-of-sight reading. RFID readers can interrogate tags at greater distances, faster and concurrently. Furthermore, one of the most important advantages of RFID technology is that tags have read/write capability, allowing stored information to be altered dynamically.

To promote the adoption of RFID technology and support interoperability, EPCGlobal [EPC Global] has ratified the EPC Class 1 Gen 2 (EPCGen2) standard for RFID deployments. This defines a platform for RFID protocol interoperability, and supports basic reliability guarantees, provided by an on-chip 16-bit pseudo-random number generator (PRNG) and a 16-bit Cyclic Redundancy Code (CRC16). The EPCGen2 standard is designed to strike a balance between cost and functionality, with less attention paid to security.

Several RFID authentication protocols that address security issues have been proposed in the literature (we refer the reader to a comprehensive repository available online at [Avoine 2010]). Cryptography especially designed for constrained devices is called lightweight, or low-cost, cryptography. Most lightweight protocols use hash functions [Sharma et al. 2003; Ohkubo et al. 2003; Henrici and Müller 2004;

[1]Research partly supported by the Spanish Ministry of Science and Innovation and the European FEDER Funds, under Project TIN 2008-02236/TSI.

Avoine and Oechslin 2005; Dimitriou 2006; Molnar et al. 2006], which are beyond the capability of low cost tags and not supported by EPCGen2. Some protocols use pseudo-random functions [Burmester et al. 2006; van Le et al. 2007; Burmester and de Medeiros 2008], or PRNGs (as in [Burmester and de Medeiros 2008; Eun Young Choi and Lim 2008]), mechanisms that are supported by EPCGen2, but are not optimized for EPCGen2 compliance. Thus, new lower complexity (*flyweight*) protocols that are suitable for EPCGen2 platforms are needed.

Some researchers have adopted a systematic approach designed to capture specific security requirements by using *privacy* models (*e.g.,* [Paise and Vaudenay 2008; Juels and Weis 2009; Michahelles et al. 2007; Avoine et al. 2007], *computational* models (*e.g.,* [Vaudenay 2007]), or *symbolic* models (*e.g.,* [Arapinis et al. 2008]). In this article we propose to use a formal specifications based framework that captures these models and addresses composability issues. This extends earlier work presented in [Burmester and de Medeiros 2009] to capture more general functionalities, such as refreshment and backward security, appropriate for lightweight RFID deployments. There is comparatively little work on RFID protocols in this framework, see *e.g.,* [Burmester et al. 2006; Burmester et al. 2006; van Le et al. 2007; Burmester et al. 2008a; 2008b; Burmester and de Medeiros 2009; Burmester et al. 2009; Burmester et al. 2009].

Our main contribution in this article is to present a novel low cost lightweight RFID protocol that supports mutual authentication with *forward* and *backward* security [Barak and Halevi 2005]. The protocol is optimistic with constant lookup time, and can be implemented on an EPCGen2 platform. Authentication is achieved by exchanging a few numbers (3 or 5) drawn from the PRNG that is shared with the back-end Server. Forward security protects *past* tag interrogations from being linked to a captured tag. Tags are not tamper-resistant, and therefore the adversary can access the private data of a captured tag. Backward security protects *future* tag interrogations from traffic analysis (correlation) attacks in which the adversary uses information leaked by tags to determine their inner state. Such attacks exploit the fact that the state of lightweight tags has low entropy. An important feature of this protocol is that RFID tags can pre-compute their response to Server challenges, and therefore the Server can detect online man-in-the-middle relay attacks by controlling the round-trip time of a challenge-response.

We then extend the universally composable (UC) security framework for RFID systems presented recently in this journal [Burmester et al. 2009], to capture lightweight-to-flyweight RFID applications, and in particular forward and backward security with refreshment. We conclude by showing that our protocol UC-realizes mutual authentication and session unlinkability with forward and backward security. We note that UC-security supports modular deployments, a feature essential for most ubiquitous applications.

Our contributions

—An analysis of recently proposed EPCGen2 protocols (Section 4).

—A Flyweight RFID protocol that provides optimistic mutual authentication with session unlinkability which extends work in [Burmester et al. 2009; Burmester and Munilla 2009] (Section 5).

—A tag refreshment mechanism, that extends the functionality of the Flyweight protocol to capture forward and backward security (Section 6).

—An implementation that addresses online relay attacks (Section 7).

—An implementation that secures the EPCGen2 Inventory protocol (Section 8).

—A UC framework that adapts the model in [Burmester and de Medeiros 2009] to capture availability,[2] mutual authentication and session unlinkability with forward and backward security (Section 9).

—A security proof and security reductions (Section 10).

1.1 A motivating paradigm

Alice wants to purchase an RFID card. Two versions are available: A $1 card for 10 interrogations and a $10 card for unlimited interrogations. She purchases the former. For her money she gets:

Availability. The card uses five numbers drawn from an on-chip PRNG to authenticate Alice. Up to 50 numbers can be drawn before correlation attacks become an issue: she is allowed 10 sessions.

Session unlinkability. The adversary cannot link sessions separated by an authorized interrogation.

The card prevents Steve, the stalker, from stealing her numbers. However Mark, the man-in-the-middle, has found a way to deplete the card. Alice gets only two authentications. She does not want to break-up with Mark so she buys the $10 Card. This time after two authentications—Mark is at it again, *Voila!* the card morphs into a brand new RFID card. This happens over and over again, whenever the card is depleted, even when Mark causes it and Alice is not in the range of an authorized reader ("quantum refreshing"). Steve and Mark give up. Mischief doesn't work. However Alice is now concerned about using her BlackBerry while getting authenticated (she is obsessed with multitasking). Ran, the analyst, assures her that the card uses a protocol that remains secure when composed with other protocols.

2. RFID DEPLOYMENTS

An RFID deployment involves tags, readers and a backend Server. Tags are wireless transponders that typically have no power of their own and respond only when they are in an electromagnetic field, while readers are transceivers that generate such fields. Tags are physically constrained devices that *cannot execute concurrently*—this will make our analysis in Section 9 much simpler. Readers implement a radio interface to tags and a high level interface to a backend Server. The Server is a trusted entity that processes private tag data. Readers do not store locally any private data and the channels that link them to the Server are assumed to be secure—hardware constraints are not so tight here, and common security protocols can be used (SSL/TLS).

[2]Protocols that employ shared security mechanisms may be subject to de-synchronization attacks.

2.1 Threats and Attacks

There are several general types of adversarial attacks on RFID deployments. Below we list the most important ones.

(1) *Tag disabling (an availability attack)*: The adversary causes tags to assume a state from which they can no longer function.
(2) *Tag cloning (an integrity attack)*: The adversary captures the identifying information of a tag.
(3) *Tag tracking (privacy)*: The adversary traces tags from their protocol flows.
(4) *Replay (integrity)*: The adversary uses the tag's response to a reader's challenge to impersonate the tag.
(5) *Man-in-the-middle offline attacks (integrity)*: The adversary interposes between a tag and a reader and exchanges their (possibly modified) messages.

There are also attacks on RFID systems that are usually excluded from the security model used, such as:

Online man-in-the-middle relay attacks [Bengio et al. 1991; Kim et al. 2008]: these are similar to the offline attacks above, with the exception that the adversary relays messages online.

Side Channel and *Power Analysis* [Mangard et al. 2007] attacks: The adversary exploits information gained by the physical implementation of protocols.

Such attacks are usually prevented by using "out of system" protection mechanisms.

In this paper we are concerned with attacks that target low cost RFID tags, in particular attacks that: exhaust the states of a tag, link tag interrogations, disambiguate *past* tag interrogations of a corrupted tag, disambiguate *future* tag interrogations of a compromised tag and, online relay attacks.

2.2 Priorities, Constraints and Optimizations

In the context of RFID applications, nearly every factor having impact on tag resources and capabilities is important. Apart from this, with EPCGen2 compliant systems, one must also take into account the execution time of the protocol: for many applications the number of tags identified per second is crucial (*e.g.,* in supply chains). Thus, we aim to *minimize* requirements for: (*i*) non-volatile RAM on the tag, (*ii*) tag code (gate count) complexity, (*iii*) tag computation requirements, (*iv*) tag turn-around-time, (*v*) the number of rounds in reader-tag interactions, (*vi*) the message size in reader-tag interactions, (*vii*) the server real-time computation load, and (*viii*) the server storage requirements.

Finally, we observe that mechanisms such as public key cryptosystems, tamper-resistant shielding, and on-board clocks are not considered realistic for low-cost applications. Furthermore symmetric-key cryptographic systems such as hash functions or encryption schemes are beyond the capability of most lightweight applications. Even, pseudo-random functions (PRF) based on PRNG (as in [van Le et al. 2007; Burmester et al. 2009]) are too slow for EPCGen2 applications (to generate an n-bit output of a PRG by running a PRNG as in [Goldreich et al. 1986] requires $2n$ numbers to be drawn).

2.3 Design Requirements

In designing our lightweight RFID protocol, we set to achieve the following goals:

Efficiency. Protocols must be lightweight: many RFID platforms can only implement highly optimized symmetric-key cryptographic techniques. Furthermore, the overhead should be minimal, in particular when the system is not under attack—we call this *optimistic performance*. Finally the lookup time for the Server should be constant, or at most logarithmic (in the number of tags).

Availability. RFID systems are vulnerable to attacks that aim to incapacitate tags, *i.e.*, force them into a state from which they cannot recover. De-synchronization attacks target availability. Such vulnerabilities are often exacerbated by the wireless and human-imperceptible nature of RFID tags, allowing them to be manipulated at a distance by covert readers.

Mutual authentication. Client authentication is a process in which one party, the Server $\mathcal{S}$, is assured of the identity of another party, the client (a tag $\mathcal{T}$), by acquiring corroborative evidence. We have *anonymous* client authentication when the identity of $\mathcal{T}$ remains private to third parties that may eavesdrop on the communication or invoke the protocol and interact with the parties directly. We have *mutual* authentication if both $\mathcal{S}$ and $\mathcal{T}$ are authenticated. In our protocol the Server is *implicitly* authenticated: that is, the assurance for tags is only implicit.

Session unlinkability. The adversary cannot link any two interrogations of a tag if, the tag either updated its state in the first, or updated it in an intermediate interrogation.

Forward Security. Past tag outputs, prior to refreshment, cannot be disambiguated by the adversary even if the adversary can access the full internal state of the tag (the state of the tag's PRNG and its private key) after it is refreshed.

Backward Security. Future tag outputs, after refreshment, cannot be disambiguated by the adversary even if the adversary knew the state of the tag's PRNG (*e.g.* by analyzing its outputs) before it was refreshed.

Concurrent Security. RFID systems are nearly always highly concurrent (a large number of tags are interrogated concurrently [EPC Global]). It is important therefore to address security in concurrent environments where the adversary can adaptively manipulate communications.

Modularity and Re-usability. Protocols are often analyzed under the implicit assumption of operating in isolation, and therefore may fail in unexpected ways when used in combination with other protocols (for example, in [Burmester and de Medeiros 2009] a proven secure route discovery protocol becomes insecure when executed concurrently with itself). Since RFID tags are components of larger systems, it is important to require that security is preserved when the protocols are executed in arbitrary composition with other (secure) protocols. This type of security is provided by the universal composability (UC) framework.

3. THE EPCGEN2 STANDARD

EPCGen2 defines the physical and logical requirements for a passive-backscatter, Interrogator-talks-first, radio-frequency identification system operating in the 860

- 960 MHz range. The system comprises Interrogators (readers) and tags. Interrogators manage tag populations using three basic operations: *select* —choose a tag population, *inventory* —identify tags, and *access* —read from and/or write to a tag.

The Inventory Protocol has (at least) 4 passes that involve: a $Query$, a 16-bit number $RN16$, an acknowledgment $ACK(RN16)$, and $EPCdata$ (a tag's identifying

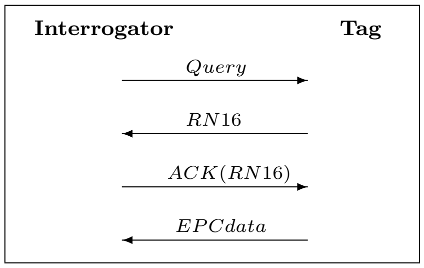

Fig. 1. The 4-pass EPCGen2 Inventory Protocol.

data)—see Figure 1. The Interrogator starts by sending a $Query$ that includes a parameter $Q \in [0:15]$. A random-slotted collision algorithm (the "Q-protocol") is used to singulate tags. Tags that receive $Query$ load a random Q-bit number into a slot counter, and decrease this counter whenever they receive the command $QueryRep$. When their counter is zeroed, tags send a random number $RN16$ to the Interrogator. When the Interrogator detects a reply from a tag, it sends an acknowledgment $ACK(RN16)$, which requests from the tag its PC (protocol control), EPC (electronic product code), and a CRC16 (cyclic redundancy code). If the Tag does not receive a valid $ACK(RN16)$ (possibly because of a collision), it transitions to its initial state and the process is repeated. Tags may also store a 32-bit Kill Password, and a 32-bit Access Password.

Tags implement a 16-bit cyclic redundancy code (CRC16) and a 16-bit random or pseudo-random number generator (PRNG). CRCs are error-detecting codes that check faults during transmission. A CRC maps arbitrary length inputs $A = (A_0, A_1, \ldots, A_{m-1})$ onto n-bit outputs as follows: first the input is represented by a polynomial $A(x) = A_0 + A_1 x + \cdots + A_{m-1}x^{m-1}$ over the finite field $GF(2)$, and then its remainder is computed modulo an appropriate generator polynomial $g(x)$ of degree n (if $m < n$, zeroes are added to make up the difference). EPCGen2 uses the CRC-CCITT generator $g(x) = x^{16} + x^{12} + x^5 + 1$, and an implementation that XORs and appends fixed bit patterns. In particular, we have:

$$CRC16(A) = [\,(A(x) + \sum_{i=m-16}^{m-1} x^i) \cdot x^{16}\,] \bmod g(x) = A(x)x^{16} \bmod g(x) + CRC16(0),$$

where $CRC16(0) = \sum_m^{m+15} x^i \bmod g(x)$ is a fixed polynomial. Since the modulo $g(x)$ operator is a homomorphism, CRC16 is semi-linear. That is, for numbers

A, B, we have:

$$CRC16(A + B) = CRC16(A) + CRC16(B) + CRC16(0).$$

Therefore the CRC16 of a sum of numbers can be computed from the CRC16s of the numbers. Consequently CRC16 by itself will not protect data against intentional alteration. Its functionality is to support error detection, in particular with respect to burst errors, not security.

A cryptographic PRNG is a deterministic function that outputs a sequence of numbers indistinguishable from random. Two components can be distinguished in a PRNG: a *state* and an algorithm *generate* (or g). To *draw* a random number from a PRNG, *state* is input to *generate*: the output is a new value for *state* and the random number. EPCGen2 does not detail how to implement this PRNG but specifies minimum randomness criteria. These guarantee a reasonable level of pseudo-randomness, except for the "collision requirement" that specifies that a drawn number RN16 should not be predictable with probability better than 0.025%, given the outcomes of prior draws. This bound is rather crude for cryptographic PRNGs: too high when only one number is drawn and too low when many numbers are drawn (*e.g.*, more than a cycle of the PRNG). In general we have to make certain that the entropy of a PRNG is sufficient and/or regularly refreshed to prevent correlation attacks. We refer the reader to [Burmester and de Medeiros 2008] for further discussion regarding these security criteria.

4. AN ANALYSIS OF RECENTLY PROPOSED EPCGEN2 PROTOCOLS

We consider five recently proposed EPCGen2 compliant protocols and show that they either fall short of their claimed security, have weaknesses that may be exploited by an adversary, or are unduly complex.

(1) The Chen-Deng protocol [Chen and Deng 2009]. This is subject to a replay attack because the flows of the Reader and tag use independent randomness (for details see [Burmester et al. 2009]).

(2) The Sun-Ting protocol Gen2$^+$ [Sun and Ting 2009]. This is also subject to a replay attack because only the tag provides randomness (for details see [Burmester et al. 2009]).

(3) The Qingling-Yiju-Yonghua protocol [Qingling et al. 2008]. This protocol uses CRC16 as a cipher. So private information can easily be manipulated, and only one eavesdropped interrogation is needed to clone a tag (for details see [Burmester et al. 2009]).

(4) Seo-Baek propose two protocols [Seo et al. 2005].

   (a) The first is subject to a replay attack (causing de-synchronization) because tag authentication does not involve any randomness from the Reader. Only one eavesdropped interrogation is needed. Again CRC16 is used as a cipher, so private information can be manipulated.

   (b) The second is also subject to a replay attack because the randomness of the flows is determined entirely by the tag. Only one previous impersonation of a Reader (sending a *query*) is needed.

(5) The Choi-Lim anti-cloning protocol [Eun Young Choi and Lim 2008]. In this protocol each tag $\mathcal{T}$ shares three private 32-bit values with the Server $\mathcal{S}$: a kill password PW_{kill}, an access password PW_{access} and a tag serial number T_{sn}. Below we describe a simplified version:

   (a) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$: Q, a query.
   $\mathcal{T}$: Select a 32-bit random number R_t and:

   (b) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$: $M_1 = R_t \oplus PW_{kill}$.
   $\mathcal{S}$: Select a 32-bit random number R_r and:

   (c) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$: $M_2 = R_r \oplus PW_{access}$ and $M_3 = RNG(R_t \oplus R_r) \oplus PW_{access}$.
   $\mathcal{T}$: get R_r from M_2. Compute $RNG(R_t \oplus R_r)$ and check M_3. If it is correct:

   (d) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$: $M_4 = RNG(RNG(R_t \oplus R_r)) \oplus T_{sn}$.
   $\mathcal{S}$: check that M_4 is correct. If it is correct, accept $\mathcal{T}$ as an authorized tag.

   This protocol has two weaknesses: (a) the Reader can be impersonated, and (b) it is subject to a *related-key attack* [Burmester and de Medeiros 2008]. For the impersonation attack, the adversary $\mathcal{A}$ first eavesdrops on an interrogation to get: Q, M_1, M_2, M_3, M_4, and then impersonates the Reader $\mathcal{R}$ as follows: when $\mathcal{T}$ sends a new M_1', $\mathcal{A}$ computes $M_2' = M_2 \oplus M_1' \oplus M_1$, and $M_3' = M_3$, and sends these to $\mathcal{T}$. These are clearly valid. Although the Choi-Lim protocol does not claim mutual authentication, if this service is not provided, it is unduly complex —see e.g. [Burmester and de Medeiros 2008].
   For the related-key attack, observe that the adversary can obtain "ciphertexts" $M4$ $(=RNG(K \oplus N_i) \oplus T_{sn})$ and "plaintexts" $M3$ $(=N_i)$ that are related by the key PW_{access} (=K). Note also that the number of the plaintexts-ciphertexts pairs is not bounded because the adversary can impersonate the Reader (attack (a)).

One may argue that because EPCGen2 supports only a very basic PRNG, any protocol that complies with this standard is potentially vulnerable, for example to ciphertext-only attacks that exhaust the range of the values of protocol flows. While this is certainly the case, such attacks may be checked by refreshing key material and/or constraining the application (e.g., the life-time of tags).

5. FLYWEIGHT RFID AUTHENTICATION

We first present a basic RFID authentication protocol which we call Flyweight. In this protocol, each tag $\mathcal{T}$ shares with the backend Server $\mathcal{S}$ a (loosely) synchronized PRNG (same algorithm, key, seed), say $g_{tag} = g_{tag}(state)$. $\mathcal{T}$ is authenticated by exchanging either three (optimistic case), or five consecutive numbers drawn from g_{tag}. Five numbers are required only when the interrogation was previously interrupted (*i.e.*, when the first number was already used: *alarm* is ON). The security of the protocol is based on the fact that: (i) it is hard for the adversary to predict the next number drawn from a PRNG, and (ii) parties $\mathcal{T}$, $\mathcal{S}$ are synchronized at all times. Synchronization is guaranteed by making certain that $\mathcal{T}, \mathcal{S}$ always share at least one number. The protocol is presented Figure 2. It supports mutual authentication, a certain degree of privacy (session unlinkability), forward and

```
(1) $\mathcal{R} \rightarrow \mathcal{T}$ :      Query
  $\mathcal{T}$ : Set $alarm \leftarrow cnt$, $cnt \leftarrow 1$. Broadcast $RN_1$.
(2) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN_1$
  $\mathcal{S}$ : Check if $RN_1$ is in $DB$.
      If $RN_1 = RN_1^{cur}$ for an item in $DB$ then set $alarm' \leftarrow cnt'$, $cnt' \leftarrow 1$, and
        broadcast $RN_2$.
      Elseif $RN_1 = RN_1^{next}$ for an item in $DB$ then set $alarm' \leftarrow 0$, $update$, and
        broadcast $RN_2$.
      Else abort.
(3) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$ :   $RN_2$
  $\mathcal{T}$ : Check $RN_2$.
      If $RN_2$ is correct then draw five successive numbers from $g_{tag}$, assign them
        to the variables $RN_3, RN_4, RN_5$ (volatile), $RN_1, RN_2$, and set $cnt \leftarrow 0$.
        If $alarm = 0$ then broadcast $RN = RN_3$.
        Else broadcast $RN = RN_4$.
      Else abort.
(4) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN$
  $\mathcal{S}$ : Check the received value $RN$.
      If $RN = RN_3$ and $alarm' = 0$ then $update$ and ACCEPT the tag as the
        authorized $\mathcal{T}$.
      Elseif $RN = RN_4$ then broadcast $RN_3$, store $RN_5$ and $update$.
      Else abort.
(5) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$ :   $RN_3$
  $\mathcal{T}$: Check $RN_3$.
      If $RN_3$ is correct and $alarm = 1$ then broadcast $RN_5$.
      Else abort.
(6) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN_5$
  $\mathcal{S}$ : Check $RN_5$.
      If $RN_5$ is correct then $update$ and ACCEPT the tag as the authorized $\mathcal{T}$.
      Else abort.
```

Fig. 2. The basic Flyweight RFID protocol.

backward security (the resilient version), and is provably secure, as we shall see in Sections 9, 10.

Each tag $\mathcal{T}$ stores in non-volatile memory two numbers, $g_{tag}(state)$ (the current state), a refresh key K, and a 1-bit flag *cnt*: $(RN_1, RN_2, g_{tag}(state), K, cnt)$. The Server $\mathcal{S}$ stores in a database DB for each $\mathcal{T}$ an ordered list containing:

—six numbers $(RN_1^{cur}, RN_1^{next}, RN_2, RN_3, RN_4, RN_5)$,
—a tag identifier ID_{tag}, $g_{tag}(state)$, the refresh key K, and
—a 1-bit flag cnt'.

The lists in DB are doubly indexed by RN_1^{cur} and RN_1^{next} respectively. To initialize the values of its variables, $\mathcal{T}$ draws two successive values RN_1, RN_2 from $g_{tag}(state)$ and sets $cnt \leftarrow 0$. $\mathcal{S}$ draws six successive numbers from the PRNG of each tag and assigns their values to the variables in the lists of the tags:

$$RN_1^{cur}, RN_2, RN_3, RN_4, RN_5, RN_1^{next} \text{ (in this order)},$$

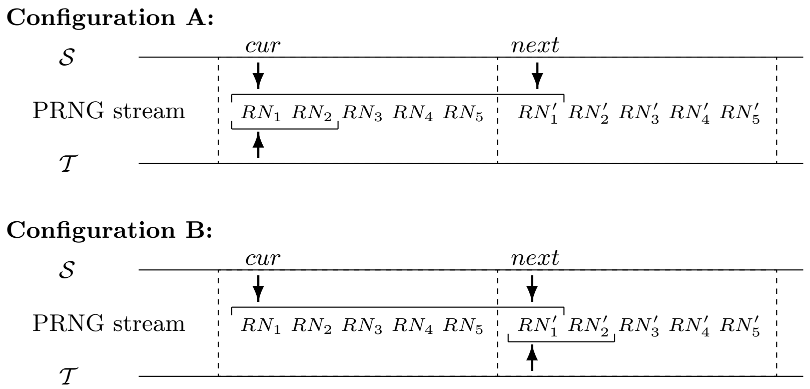
Configuration A:
cur
next
$\mathcal{S}$
PRNG stream
RN_1 RN_2 RN_3 RN_4 RN_5
RN'_1 RN'_2 RN'_3 RN'_4 RN'_5
$\mathcal{T}$
Configuration B:
cur
next
$\mathcal{S}$
PRNG stream
RN_1 RN_2 RN_3 RN_4 RN_5
RN'_1 RN'_2 RN'_3 RN'_4 RN'_5
$\mathcal{T}$

Fig. 3. Synchronizing the pseudo-random streams of $\mathcal{S}$ and $\mathcal{T}$.

and sets *cnt'* $\leftarrow 0$. To update a list, $\mathcal{S}$ uses the function *update* for which: $RN_1^{cur} \leftarrow RN_1^{next}$, the five values $RN_2, RN_3, RN_4, RN_5, RN_1^{next}$ are updated by drawing new numbers from $g_{tag}(state)$, and *cnt'* $\leftarrow 0$.

Synchronization. At all times the Server $\mathcal{S}$ shares with each tag $\mathcal{T}$ at least one number: either $RN_1 = RN_1^{cur}$ or $RN_1 = RN_1^{next}$. We distinguish two cases identified by Configuration A and Configuration B in Figure 3. $\mathcal{S}$, $\mathcal{T}$ each use a block of five successive numbers from their pseudo-random stream (drawn from their PRNGs) for each session. Configuration A describes the normal state, when the previous flow was not interrupted: in this case $\mathcal{S}$, $\mathcal{T}$ use the same block. When $\mathcal{T}$ receives RN_2, it sends RN_3 or RN_4 and moves to the next block. If the message of $\mathcal{T}$ (RN_3 or RN_4) is interrupted, then Configuration B will be the initial state for the next session. Otherwise, $\mathcal{S}$ receives $\mathcal{T}$'s message and also moves ahead to the next block, returning to Configuration A. When the initial state is described by Configuration B then $\mathcal{S}$ receives RN_1^{next} and will advance to the next block. It must be noted that the synchronization process is independent of the authentication process: *i.e.,* the parties can advance along the stream and get synchronized even when the authentication was not completed successfully. The adversary may try to de-synchronize $\mathcal{T}$ by challenging it with a *Query*, or the number $RN2$ obtained by using a man-in-the-middle attack on $\mathcal{S}$, $\mathcal{T}$. In the first case $\mathcal{T}$ will not update its stored values, so it will share $RN_1 = RN_1^{cur}$ with $\mathcal{S}$. In the second, $\mathcal{T}$ updates its values and will share $RN_1 = RN_1^{next}$ with $\mathcal{S}$ (RN'_1 in Figure 3). The protocol prevents any further updating by $\mathcal{T}$ before $\mathcal{S}$ does: $\mathcal{T}$ can only update its values when it is prompted by RN_2 (RN'_2 in Figure 3).

Optimistic behavior. When the adversary is passive then only three numbers have to be exchanged to authenticate a tag $\mathcal{T}$. If an active adversary tries to replay flows, this will cause $\mathcal{T}$ to activate *alarm*, and two additional numbers will be needed (Pass 5 and Pass 6). Observe that the numbers RN_3, RN_4 and RN_5 are always fresh (never sent more than once), because at this point $\mathcal{S}$ and $\mathcal{T}$ have already updated their pseudo-random values for the next interrogation.

Constant Lookup time. The Server needs to perform at most two lookups in the database DB (for RN_1^{cur} and RN_1^{next}) to identify $\mathcal{T}$. The cost of a simple key lookup in DB using Linear Hashing is $O(1)$ [Zhang et al. 2009].

Timers. We have not included timers to simplify the presentation. However in any implementation these are needed to close sessions. Parties should abort if no response is received after sending a challenge within a certain time T_{abort}. If the timers are precise enough, they can be used to thwart online man-in-the-middle relay attacks (*cf.* Section 2.1). The more accurate T_{abort} is, the harder the attacks become [Munilla et al. 2006]. Naturally, an active attack that involves relaying flows between protocol parties faster than T_{abort} will succeed. In Section 7 we will explain how to deal with these attacks in more detail.

Implementation complexity. There are several efficient implementations of PRNGs appropriate for lightweight RFID applications. The shrinking generator of Coppersmith, Krawczyk and Mansour [Coppersmith et al. 1994], which is based on linear feedback shift registers, has been estimated to require only 1435 logic gates, 517 clock cycles and $64B$ memory (clock frequency $100KHz$), and achieves 128 bit security [Lee and Hong 2006].

More recently, a hardware implementation LAMED-EPC [Peris-Lopez et al. 2009] specifically tailored for EPCGen2 applications has been proposed. This is estimated to require 1566 logic gates, 194 clock cycles ($100KHz$), $64B$ memory. It is EPCGen2 compliant.

Note that PRNGs are typically a few orders faster than pseudo-random functions (PRFs) and therefore RFID implementations that use PRNGs will run faster (PRFs can be generated from PRNGs [Goldreich et al. 1986]).

Session unlinkability. The only instances in which the adversary can link sessions are those in which the tag is prevented from accessing an authorized reader. In such cases the tag outputs the same number RN_1 each time, so these flows can be linked. On receiving a response RN_2 from the reader the tag will update its stored values, and its flows become unlinkable.

6. RESILIENT FLYWEIGHT RFID PROTOCOL

6.1 Refreshing Random Number Generators

Our protocol uses (loosely) synchronized PRNGs that can be refreshed. PRNGs are refreshed to ensure resilience against traffic analysis attacks that exploit the correlation between successive numbers drawn from a PRNG (*state entropy leakage*). That is, to ensure that the adversary cannot predict with probability better than a certain threshold p_0:

(1) the next number drawn (*e.g.,* by using an exhaustive analysis of all possible states that produce the tag's output), and/or

(2) the state of the PRNG,

until the tag is next refreshed. For a detailed discussion on security issues of PRNGs see [Barak and Halevi 2005; Kelsey et al. 1998]. Furthermore, as we shall see, refreshing a PRNG will restrict the impact of a compromised *state*.[3]

Refreshing a PRNG involves updating *state* with fresh (high entropy) randomness—see Figure 4. In our protocol this randomness is provided by the Server when needed: *e.g.*, when the probability that the state of the PRNG of the tag is compromised is higher than a certain threshold (this will depend on the specific features of the implemented PRNG).

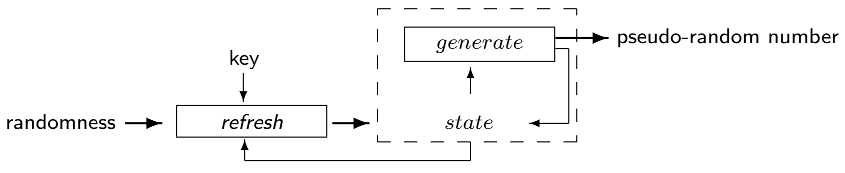

Fig. 4. Refreshing the *state* of a PRNG.

To refresh the *state* of a PRNG, we combine it with randomness and input this to a keyed *refresh* function to get:

$$state^{ref} = refresh(K; R, state),$$

where K is the refresh key and R randomness. In the following definition we distinguish between corrupted and compromised tags. A tag $\mathcal{T}$ is *corrupted* if the adversary has access to its PRNG (its state) and the refresh key K; $\mathcal{T}$ is *compromised* if the adversary has access only to its PRNG (can predict the numbers drawn from its PRNG).

Definition 6.1. Let $\mathcal{A}$ be the adversary, $\mathcal{T}$ a tag that is refreshed and H a history of tag interrogations captured by $\mathcal{A}$.

—*Forward security.* Suppose that $\mathcal{T}$ gets corrupted after it is refreshed: then the adversary cannot disambiguate the outputs of $\mathcal{T}$ in H captured prior to refreshment.

—*Backward security.* Suppose that $\mathcal{T}$ gets refreshed after it was compromised: then the adversary cannot disambiguate the outputs of $\mathcal{T}$ in H captured after refreshment.

Definition 6.2. *Resiliency*: A *refresh* function supports *resiliency* for an RFID system if it guarantees forward and backward security.

6.2 Adding resiliency to the Flyweight protocol

We now describe the modifications to the Flyweight protocol needed to refresh the PRNGs of tags. For each tag the Server uses a 1-bit trigger *refresh* and stores additionally five numbers:

—A high entropy random number R

[3]The term "compromised state" is used here in a broad sense: it specifies the information captured by the adversary in a correlation attack used to predict the next number drawn from a PRNG. The prediction is probabilistic, not necessarily deterministic.

—Numbers NA (start of refresh), NB (end of refresh)

—Numbers $RN5'$ (a message authentication code for R) and $N0$ (initial point)

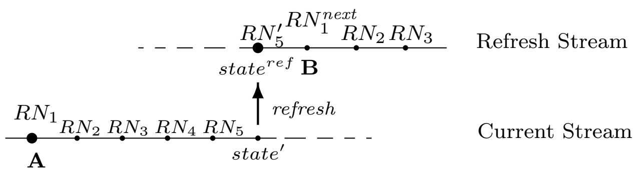

Fig. 5. Transition from the Current to the Refresh Stream.

In Figure 5 we illustrate the process of refreshing a stream generated by a shared PRNG. **A** and **B** mark the start and end of the refreshing.

The modifications are presented in Figure 6. Only Flow 3 is affected, with the reader sending two numbers R, RN_5' instead of R_2. *Normal execution* indicates that the pass is executed as in the basic protocol, presented in Figure 2.

To refresh a tag the Server generates a random number R, and sets: *refresh* ON, the initial point $N0 \leftarrow RN_1^{cur}$, the start refresh number $NA \leftarrow RN_1^{next}$, and the end refresh number NB and RN_5' to null. When the start number is received ($RN_1 = NA$), the server computes RN_5' and $NB = RN_1^{next}$ on the Refresh Stream (Figure 5), and broadcasts R, RN_5'. If NA is received again, the Server broadcast the same R, RN_5' (never RN_2). The number RN_5' authenticates both the refresh session and the random number R. If the tag gets R, RN_5' (a message format with two numbers) then it draws numbers from $g_{tag}(state)$ to get $state'$, which it refreshes to get its own evaluation of RN_5'. If there is a match the protocol continues normally. In the following session all numbers drawn will be on the Refresh Stream ($RN_1 = NB$). When NB is received, both tag and Server have refreshed and the process has finished. The tag's computations for checking the value of RN_5' are done in volatile memory: the tag must keep the original *state* of its PRNG in non-volatile memory in case there is a mismatch, so that it can reset to its initial state.

Synchronization. In the resilient Flyweight protocol at all times the Server $\mathcal{S}$ shares with each tag $\mathcal{T}$ at least one number: either NA or NB. The adversary may try to de-synchronize $\mathcal{T}$ by trying to force it to advance on the current stream while the Server $\mathcal{S}$ updates and advances on the refreshed stream—Figure 5. However, this is not possible since $\mathcal{T}$ would need RN_2 to advance on the current stream and this number is never sent by $\mathcal{S}$, which will repeatedly send R, $RN5'$ (Flow 3) until it gets the correct RN (Flow 4) or NB (Flow 2). If the tag updates—and sends RN or NB, this is because it has checked R, $RN5'$ and refreshed its PRNG properly.

Compromising PRNGs. PRNGs are refreshed to prevent the adversary from compromising their *state*, and predict each bit of the next number drawn with probability significantly better than 0.5. If the adversary succeeds in compromising the *state* of the PRNG of a tag before it gets refreshed then it can impersonate that tag, and/or de-synchronize it (by getting the Server to *update* through interrogation). Typically such attacks exploit the inadequate frequency of refreshing,

```
$\mathcal{S}$ : set $refresh$ ON, generate a random number $R$, assign $N0 \leftarrow RN_1^{cur}$, $NA \leftarrow RN_1^{next}$,
        $NB \leftarrow \perp$ and $RN_5' \leftarrow \perp$

  (1) $\mathcal{R} \rightarrow \mathcal{T}$ :   Query
    $\mathcal{T}$ : $Normal\ execution.$
  (2) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN_1$
    $\mathcal{S}$ when $refresh$ ON: Check if $RN_1$ is in $DB$
        If $RN_1 = N0$ then $Normal\ execution$
        Elseif $RN_1 = NB$ then set $refresh$ OFF and $Normal\ execution$
        Elseif $RN_1 = NA$ then,
            If $RN_1 = RN_1^{next}$ then update
            If $RN5' = \perp$ then set $state^{ref} \leftarrow refresh(K; R, state')$. Draw two numbers
                and assign to $RN5'$ and $RN_1^{next}$.
            Set $alarm' \leftarrow cnt', cnt' \leftarrow 1$. Broadcast $R$, $RN_5'$ .
        Else abort
  (3) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$ :   $R$, $RN_5'$   (Refresh Pass)
    $\mathcal{T}$ : Check the format of the received message.
        If it corresponds to “refresh” (two numbers) then,
            Store the current $state$ of $g_{tag}$. Draw 3 numbers from $g_{tag}(state)$ and assign
            their values to $RN_3, RN_4, RN_5$ (volatile). Draw an extra number to get $state'$,
            and set $state^{ref} \leftarrow refresh(K; R, state')$.
            Draw one number from $g_{tag}(state^{ref})$.
            If it is $RN_5'$ then draw two more numbers and assign their values to $RN_1, RN_2$.
                If $alarm = 0$ then broadcast $RN = RN_3$.
                Else broadcast $RN = RN_4$.
            Else reset the state of $g_{tag}$ and abort.
        Else $Normal\ execution.$
  (4) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN$   ($RN_3$ or $RN_4$)
    $\mathcal{S}$ : $Normal\ execution.$
  (5) $\mathcal{S} \Rightarrow \mathcal{R} \rightarrow \mathcal{T}$ :   $RN_3$
    $\mathcal{T}$: $Normal\ execution.$
  (6) $\mathcal{T} \rightarrow \mathcal{R} \Rightarrow \mathcal{S}$ :   $RN_5$
    $\mathcal{T}$ : $Normal\ execution.$
```

Fig. 6. The resilient Flyweight RFID protocol.

or the low entropy of the seed of the PRNG. They cannot be considered attacks on the Flyweight protocol itself, which is proven secure in Section 9, but on the security parameters used. However, the implementation of refresh (if this is done properly) restricts the impact of such impersonation attacks until the next refreshing. The adversary cannot compute the refreshed *state* without knowing the refresh key K. This can be used by the Server $\mathcal{S}$ to revive a genuine tag which now is desynchronized (a *zombie* tag). $\mathcal{S}$ accepts previously used (since the last refreshing) values (RN_1), but it forces the tag $\mathcal{T}$ to refresh again at this point. Only if $\mathcal{T}$ is genuine and knows K, will it be able to refresh *state* correctly.

Another possible way to refresh the PRNG of a tag with entropy from the Server involves flipping the order of the numbers drawn, *e.g.*, RN_2 and RN_3, so that one bit of the state of the tag (determined by a counter) is refreshed. This would support resilience against correlation attacks if the information leaked when five numbers are drawn from a PRNG is no more than one bit. We shall discuss the security of our protocol in Section 9.

7. ONLINE RELAY ATTACKS

Distance bounding protocols based on round-trip delay measurements are the main defense against attacks related to proximity verification. These estimate the propagation time as accurately as possible so as to determine the distance between the reader and tag. Determining the processing time is essential in order to isolate the propagation component from the overall measured time, and therefore variable processing times constitute a major problem for distance bounding. Apart from being invariable, the processing time must be as short as possible since an adversary could overclock the tag to absorb the delay introduced by his own devices. Thus, the Flyweight protocol with its fixed nearly-zero processing delay is particularly suitable to protect against online man-in-the-middle relay attacks (Section 2.1). This is because in the Flyweight protocol every tag pre-computes its response (RN/RN_5 drawn from its PRNG) to the challenge of the reader (RN_2/RN_3).

To estimate the round-trip time of a challenge-response we use *temporal leashes* [Hu et al. 2006]. The reader must have an accurate clock, but there is no need for the tags to have clocks (depending on the implementation we may require the Server to have a clock that is synchronized with the clock(s) of the reader(s)). Let δ_0 be a temporal bound calculated using a distance bound (the allowable reader-tag broadcast range), the propagation speed of the wireless medium (*i.e.,* the speed of light) and the tag processing time (which includes the time taken to detect the challenge and transmit the response). If the challenge (RN_2/RN_3) of the reader is sent at time t_1 and the response of the tag (RN/RN_5) is received at time t_2, then the reader will only accept it when $t_2 - t_1 \leq \delta_0$. If the delay introduced by the adversary's devices is greater than δ_0 then online relay attacks will be prevented: *i.e.,* the adversary will not be able to relay the messages without being detected.

This simple way to address online relay attacks and the constant lookup time, highlight the extended functionality that is provided by sharing a synchronized RFID stream as opposed to sharing a private number (key)—captured by "quantum refreshing" in the introductory motivating paradigm.

8. AN EPCGEN2 IMPLEMENTATION

The EPCGen2 Inventory protocol has 4 passes for identification (acknowledged state): $Query$, $RN16$, $Ack(RN16)$ and $EPCdata$ (Section 2, Figure 1). To enable mutual authentication we replace $RN16$ by RN_1, $Ack(RN16)$ by RN_2 and $EPCdata$ by RN_3 (optimistic case). We illustrate the modifications in Figure 7. On the left is the 4-pass EPCGen2 Inventory protocol, while on the right is the proposed Flyweight Inventory protocol. Note that the latter requires two additional passes for secure mutual authentication when the adversary is active (RN_1 has been used—$alarm$ is ON).

To ensure that it is hard to find the state of an EPCGen2 PRNG by using an exhaustive search over all possible state values that produce a given sequence of numbers, the entropy of the state of PRNG must be sufficiently large. If a 32-bit

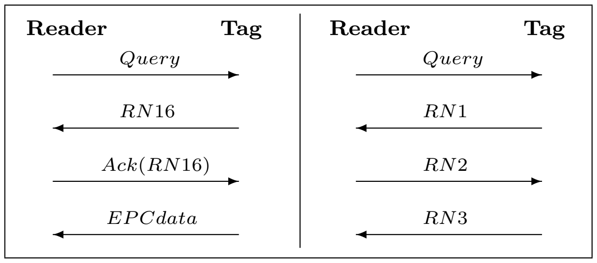

Fig. 7. The 4-pass EPCGen2 Inventory (left) and the proposed Flyweight Inventory.

state[4] with refreshment provides adequate security then we may use the following simple implementation:

$$refresh(K; R, state) = g_{tag}(K \oplus R \oplus state),$$

where R is a 32-bit random number and K the 32-bit Access Password. At this point one may think that if the PRNG is used as a pseudo-random function (PRF), then many other solutions (apart from the Flyweight protocol) are feasible. However, one has to be careful: PRNGs when used as PRF may be subject to *related-key* attacks [Burmester and de Medeiros 2008]. That is, if g is a PRNG and k a key then there are no guarantees that $g(k \oplus x)$ (or $g(k, x)$) is a secure message authentication code (MAC) for x (at least, not until proven). In a *related-key* attack, the adversary uses values drawn from $g(k, \cdot)$ (or $g(\cdot)$), whose keys are related (in this case, the same), to infer information about the next numbers drawn. If the adversary can choose the values of x then the problem is far worse, because the adversary can perform adaptive attacks. Most protocols that use a PRNG to generate message authentication codes (*e.g.,* [Choi et al. 2009; Huang and Ku 2009]) are subject to such attacks. This problem is prevented in the Flyweight protocol by using synchronized PRNGs, since the value of *state* is not known by the adversary, and changes dynamically. To conclude we note that one may use a provable secure (but slower) alternative,

$$PRF_{tag}(K \oplus R \oplus state),$$

where PRF_{tag} is the PRF defined by g_{tag} [Goldreich et al. 1986].

8.1 Collisions

We have collisions during Inventory and in DB. With Inventory collisions, several tags in the operating range of the reader respond to the same Query. This is solved in EPCGen2 by using the Q-protocol in which a random Q-bit number is loaded into a slot counter and decreased with each interrogation (Section 3): the tags respond when it is zeroed. In our case we use the bits of RN_1 instead of Q ($alarm$ is activated if these bits are used more than once)

With collisions in DB several tags share the same RN_1. In this case the Server receives RN_1 but does not know which of the RN_2's in DB must be sent because

[4]This does not affect the length of the outputs, which can still be of 16 bits.

several numbers are possible. The easiest way to deal with this, is to modify the protocol and have a backup RN_1^{backup}. In this case the parties use blocks of six pseudo-random numbers. Then, when the Server detects a collision, it sends a new command $QueryB$, requesting the tag to send RN_1^{backup}. By using extra numbers, the probability of collision can be reduced as much as needed.

However, the protocol can be modified to deal with collisions without requiring an extra number RN_1^{backup}. We propose the following solution. Suppose there is a collision in DB for RN_1. The Server tries to identify $\mathcal{T}$ by sending RN_2's which were previously used (for which $alarm = 1$). If the tag responds with RN_4, then $\mathcal{T}$ is identified, and the six-pass protocol is executed. If RN_2 was not previously used by $\mathcal{T}$, then the Server sends RN_5 to inform the tag $\mathcal{T}$ of the collision. When $\mathcal{T}$ gets RN_5 in the second pass, it exchanges RN_2 with RN_5, and proceeds normally. Tags for which RN_5 was sent prior to identification get marked by the Server and their identification must be performed without using such RN_5: that is, with four passes in the optimistic case or in a new updated session.

Finally, we could have the unlikely event when several tags that share the same RN_1 are interrogated simultaneously (simultaneous collision in the Inventory and in DB). In this case the Server would not detect the collision during the Inventory (constructive interference), and the Server would deal with it as a collision in DB. The Server sends a previously used RN_2 or RN_5. The tag (among the ones present) whose value coincides with this (RN_2 or RN_5) will answer and will be identified. The remaining tags get identified one-at-a-time in the same way (with new $Queries$).

9. SECURITY

Our formal security specifications are: *mutual authentication,* and *session unlinkability* with *forward* and *backward* security. Since RFIDs are often used as components of more complex systems, we focus on security frameworks that support Universal Composability (UC). The choice of cryptographic primitives to implement the protocols must take into consideration: (*i*) the need for computationally lightweight solutions that adhere to the hardware-imposed constraints of the platform, and (*ii*) scalability, when the number of devices is large. We will use the security framework proposed in [Burmester et al. 2009], which we extend to accommodate our particular specifications.

We adopt the Byzantine threat model. All parties including the adversary $\mathcal{A}$ are modeled as probabilistic polynomial-time Turing machines (PPTs). $\mathcal{A}$ controls the delivery schedule of all communication channels, and may eavesdrop into, or modify, their contents and may also initiate new communication channels and directly interact with honest parties. For convenience, in our proofs below, we will identify the readers with the Server.

9.1 The security framework

The universal composability (UC) framework specifies a particular approach to security proofs for protocols π, and guarantees that proofs that follow this approach remain valid if π is, say, composed with other protocols (modularity) and/or under arbitrary concurrent protocol executions (including with itself). The UC framework defines a *real world simulation*, an *ideal world simulation*, a *simulator* $\mathcal{S}_{sim}$ that translates runs of π from the real world to the ideal world, and an interactive *environment* $\mathcal{Z}$, a PPT, that captures whatever is external to the current protocol execution. The components of a UC formalization are:

(1) A *mathematical model* of real executions of protocol π in which the honest parties execute as specified, whereas adversarial parties can deviate from π arbitrarily. These are controlled by the adversary $\mathcal{A}$ that has full knowledge of the state of adversarial parties, and can arbitrarily schedule the communication channels and activation periods of all parties, and interact with $\mathcal{Z}$ in arbitrary ways.

(2) An *idealized model* of executions, where the security properties of protocol π depend on the behavior of a *trusted functionality* $\mathcal{F}_\pi$. $\mathcal{F}_\pi$ controls the ideal world adversary $\widehat{\mathcal{A}}$ so that it reproduces as faithfully as possible the behavior of $\mathcal{A}$.

(3) A *proof* that, for each $\mathcal{A}$ there is a simulator $\mathcal{S}_{sim}$ that translates real world protocol runs of π in the presence of $\mathcal{A}$ into ideal world runs of π in the presence of $\widehat{\mathcal{A}}$ such that, no environment $\mathcal{Z}$ can distinguish whether $\mathcal{A}$ is communicating with a instance of π in the real world or $\widehat{\mathcal{A}}$ is communicating with $\mathcal{F}_\pi$ in the ideal world.

In the UC framework $\mathcal{Z}$ is the first party to be activated. It instantiates the protocol parties and the adversary $\mathcal{A}$.

9.2 Mutual authentication with session unlinkability

Mutual authentication with session unlinkability in the UC framework is captured by the parties (the Server and tags) having access to an ideal functionality which we denote by $\mathcal{F}_{asu}$. $\mathcal{F}_{asu}$ formally defines the security specifications for, availability, mutual authentication and session unlinkability, in protocols for which the Server and tag share a (loosely) synchronized PRNG. It is presented in Figure 8. $\mathcal{F}_{asu}$ specifies protocols for which the tags determine the interrogation subsession. Below we describe the basic components and attributes of the ideal world simulation.

SESSIONS. A single session spans the entire lifetime of our system. It consists of several concurrent subsessions which are INITIATEd by the protocol parties, which in turn get initiated by the environment $\mathcal{Z}$. While the Server and tags initiate subsessions, the adversary controls the concurrency and interaction between subsessions. All parties involved in a subsession of the authentication scheme are given a unique identifier *sid* by $\mathcal{Z}$ (*sid* includes identifying data of tags, private data etc).

CONCURRENCY. Tags are constrained devices that cannot execute concurrently. In the ideal world this is captured by restricting tags to one subsession identifier s at a time. The adversary cannot INITIATE the same tag concurrently.

AVAILABILITY. In the real world this requires that a tag is always available for interrogation. In the ideal world this is captured by assigning to each INITIATEd tag a subsession identifier s and making it available for interrogation by invoking commands UPDATE(s), ACCEPT(s), IMPERSONATE(s) or CORRUPT(s).

MUTUAL AUTHENTICATION. Successful authentication in the real world is the re-

Functionality $\mathcal{F}_{asu}$

$\mathcal{F}_{asu}$ only accepts commands with the session identifier *sid*

Upon input INITIATE at `server`. Record $\texttt{init}(server)$ if there is no such record and output $\texttt{init}(server)$ to the adversary.

Upon input INITIATE at `tag`. If `tag` is corrupted then ignore. Else, generate a new subsession identifier s. If there is no record $\texttt{init}(s', \texttt{tag})$ or $\texttt{update}(s', \texttt{tag})$ then record $\texttt{init}(s, \texttt{tag})$ and $session\_id(\texttt{tag}) = (s)$ (a list). Else discard the record, record $\texttt{init}(s, \texttt{tag})$, append s to $session\_id(\texttt{tag})$ and output $session\_id(\texttt{tag})$ to the adversary.

Upon request UPDATE(s) from the adversary. SFunctionalIf there is a record $\texttt{update}(\texttt{tag})$ then ignore. Else if there is a record $\texttt{init}(s, \texttt{tag})$ then remove it, generate a record $\texttt{update}(s, \texttt{tag})$, discard all entries from $session\_id(\texttt{tag})$ and record $\texttt{update}(\texttt{tag})$ at `tag`.

Upon request ACCEPT(s) from the adversary. If there is a record $\texttt{update}(s, \texttt{tag})$ then remove it and record $\texttt{accept}(\texttt{tag})$ at `server`.

Upon request IMPERSONATE(s) from the adversary. If `tag` is corrupted then record $\texttt{accept}(\texttt{tag})$ at `server`.

Upon request CORRUPT(s) from the adversary. If there is a record $\texttt{init}(s, \texttt{tag})$ or $\texttt{update}(s, \texttt{tag})$, then mark `tag` as corrupted and remove $state(\texttt{tag})$.

Fig. 8. Ideal mutual authentication with session unlinkability.

sult of sharing common secrets: the Server can corroborate values produced by the tag as a function of a (loosely) synchronized shared PRNG and conversely. The choice of the tag to be authenticated is determined by $\mathcal{A}$. To guarantee that the PRNG remains synchronized, mutual authentication in the real world requires that the tag update its state. In the ideal world this is captured by invoking command ACCEPT (Item 4, Figure 8). To get a tag with subsession identifier s authenticated, command UPDATE(s) must have been invoked. The true identity of a tag is given to the `server`, but not $\mathcal{A}$. This limits $\mathcal{A}$ to invoking and scheduling the protocol at each party.

SESSION UNLINKABILITY. In the real world session unlinkability requires that given any two tag interrogations, if the tag has updated its state in the first, or in an intermediate interrogation, then the adversary cannot link these with probability better than negligible. The adversary can link interrogations that have not been updated (they have the same value) until the next successful updating. In the ideal world this is emulated by acquiring access to a list of identifiers $session\_id(\texttt{tag})$ of all preceding incomplete subsessions returned by the functionality by invoking INITIATE at the `tag` (Item 2, Figure 8). The only information revealed to the adversary by the functionality is the subsession identifiers of tags: no information regarding the tag itself is revealed. Once a tag with subsession identifier s is successfully UPDATEd in the ideal world, all earlier subsessions identifiers (in $session\_id(\texttt{tag})$) of the same tag are discarded by the functionality (Item 3, Figure 8).

TAG CORRUPTION. In the real world tags may get corrupted by the adversary, who can then access their full state. This is emulated in the ideal world by invoking command CORRUPT (Item 6, Figure 8). The functionality maintains for each `tag` a list *state*(`tag`) of all subsession records concerning `tag`. This list is discarded by the ideal functionality upon corruption of the tag when invoking command CORRUPT. Consequently in the ideal world, control of a corrupted tag is passed on to the adversary.

ACTIVATION SEQUENCE. The receiving party of any message is activated next. If no output is produced while processing an incoming message then by convention the environment $\mathcal{Z}$ is activated next.

9.3 Capturing forward and backward security

To capture tag resilience we need two more commands: COMPROMISE and REFRESH. Invoking COMPROMISE(s), where s is a subsession identifier of `tag`, results in `tag` getting marked as compromised by the functionality. COMPROMISEd tags can be successfully IMPERSONATEd by the adversary until they are REFRESHed, when they get de-synchronized—*zombie* tags. Such tags cannot be IMPERSONATEd by the adversary unless command COMPROMISE is invoked again. The functionality $\mathcal{F}_{rasu}$ is presented in Figure 9. The additional attributes are:

FORWARD SECURITY. In the real world this requires that past protocol flows, prior to refreshment, look random even if the tag gets corrupted after REFRESHment. In the ideal world this is emulated by requiring that after REFRESH(s) all entries in *session_id*(`tag`) are discarded, so past sessions look random (Item 4, Figure 9).[5]

BACKWARD SECURITY. In the real world this requires that future tag outputs, after refreshment, look random to an adversary even if the adversary can access the state of the PRNG of the tag (e.g., by analyzing its outputs) before it is refreshed. In the ideal world tag compromise is emulated by invoking command COMPROMISE. Tags that get compromised are marked by $\mathcal{F}_{asu}$ as such, and therefore can be successfully IMPERSONATEd by the adversary (Item 6, Figure 9). However after REFRESHment a compromised tag is marked zombie and cannot be IMPERSONATEd.

10. MAIN RESULTS AND PROOFS

We first consider the security of the basic protocol.

THEOREM 10.1. *The Flyweight RFID protocol realizes the UC-functionality* $\mathcal{F}_{asu}$.

PROOF. To prove that the Flyweight protocol realizes $\mathcal{F}_{asu}$ we must show that Condition 3 in Section 9.1 holds, that is, there is a simulator $\mathcal{S}_{sim}$ that translates real world protocol runs into ideal world runs such that these cannot be distinguished by $\mathcal{Z}$. Our simulator $\mathcal{S}_{sim}$:

[5]The ideal world specifications for REFRESH and UPDATE are essentially the same—except for zombie tags. In our UC framework with a PPT distinguisher (environment) $\mathcal{Z}$, this is reflected in the real world simulation. To capture resilience to entropy leakage resulting from correlation attacks on the numbers drawn from a PRNG one has to use concrete security reductions, as in Section 10.1, 10.3.

Functionality $\mathcal{F}_{rasu}$

$\mathcal{F}_{rasu}$ has session identity *sid* and only accepts commands with the same *sid*.

Upon input INITIATE at `server`. Record `init(server)` if there is no such record and output `init(server)` to the adversary.

Upon input INITIATE at `tag`. If `tag` is corrupted then ignore. Else, generate a new subsession identifier s. If there is no record `init`$(s', $`tag`$)$ or `update`$(s', $`tag`$)$ then record `init`$(s, $`tag`$)$ and *session_id*(`tag`) $= (s)$ (a list). Else discard the record, record `init`$(s, $`tag`$)$, append s to *session_id*(`tag`) and output *session_id*(`tag`) to the adversary.

Upon request UPDATE(s) from the adversary. If there is a record `update(tag)` then ignore. Else if there is a record `init`$(s, $`tag`$)$ then remove it, generate a record `update`$(s, $`tag`$)$, discard all entries in *session_id*(`tag`) and record `update(tag)` at `tag`.

Upon request input REFRESH(s) at `server`. If there is a record `update(tag)` then ignore. Else if there is a record `init`$(s, $`tag`$)$ then remove it, generate a record `update`$(s, $`tag`$)$, discard all entries in *session_id*(`tag`) and if `tag` is compromised then mark it zombie, unless it already has this designation, and record `update(tag)` at `tag`.

Upon request ACCEPT(s) from the adversary. If there is a record `update`$(s, $`tag`$)$ then remove it and record `accept(tag)` at `server`.

Upon request IMPERSONATE(s) from the adversary. If `tag` is corrupted then record `accept(tag)` at `server`. If there is a record `init`$(s, $`tag`$')$ or `update`$(s, $`tag`$')$ and `tag` is compromised, but not zombie, then remove it and record `accept(tag)` at `server`.

Upon request COMPROMISE(s) from the adversary. If there is a record `init`$(s, $`tag`$)$ or `update`$(s, $`tag`$)$ and `tag` is not compromised then mark it compromised.

Upon request CORRUPT(s) from the adversary. If there is a record `init`$(s, $`tag`$)$ or `update`$(s, $`tag`$)$, then mark `tag` as corrupted and remove *state*(`tag`).

Fig. 9. Ideal resilient authentication.

—Simulates a copy $\widehat{\mathcal{A}}$ of the adversary and copies $\widehat{\texttt{server}}$ of the Server, and $\widehat{\texttt{tag}}$ of each tag initialized by $\mathcal{Z}$, and activated by the adversary.

—Adds/removes values to/from a database $\widehat{DB}$ of $\widehat{\mathsf{server}}$ that contains persistent values of adversarially controlled tags as well as the transient values of honest tags. The simulated interactions between $\mathcal{F}_{asu}$ and $\widehat{\mathcal{A}}, \widehat{\mathsf{server}}, \widehat{\texttt{tag}}$ are detailed in Figure 10.

—Faithfully translates real world messages between $\{\mathcal{A}$, Server,tag$\}$ into their ideal world counterparts between $\{\widehat{\mathcal{A}}, \widehat{\mathsf{server}}, \widehat{\texttt{tag}}_s\}$ as specified by the Flyweight protocol. This is detailed in Figure 11. In this simulation the ideal world adversary $\widehat{\mathcal{A}}$ invokes SEND$(s, r, \widehat{\texttt{tag}})$ to send to $\widehat{\texttt{tag}}_s$ the number r, and SEND$(s, r', \widehat{\texttt{server}})$ to send to $\widehat{\texttt{server}}$ the number r'.

—Simulates the interactions with $\mathcal{Z}$, i.e., the externally visible part of the protocol. More specifically, it invokes $\mathcal{F}_{asu}$ with command ACCEPT(s) when the real world adversary forwards unmodified inputs between honest tags and the Server, and IMPERSONATE(s, tag) when the real world adversary $\mathcal{A}$ succeeds in authenticating adversarially controlled tags.

Simulated interactions by $\mathcal{S}_{sim}$, I

Upon $\texttt{init(server)}$ from $\mathcal{F}_{asu}$. Send $\texttt{init(server)}$ to $\widehat{\mathcal{A}}$.

Upon $session\_id(\texttt{tag})$ from $\mathcal{F}_{asu}$. If $session\_id(\texttt{tag}) = (s)$ then create a new tag named $\widehat{\texttt{tag}}_s$, and generate $flow(s) = (r_1, r_2, r_3, r_4, r_5)$. Give $(s, flow(s))$ to $\widehat{\texttt{tag}}_s$ and store it in $\widehat{DB}$ together with $session\_id(\texttt{tag})$ using the identity $\widehat{\texttt{tag}}_s$. Else, if s' is the first identifier in $session\_id(\texttt{tag})$ then: assign s to $\widehat{\texttt{tag}}_s = \widehat{\texttt{tag}}_{s'}$ and set $flow(s) = flow(s')$. Send $session\_id(\texttt{tag})$ to $\widehat{\mathcal{A}}$.

Upon $\texttt{update}(s)$ from $\widehat{\texttt{tag}}_s$ or $\widehat{\texttt{server}}$ in subsession s. Send UPDATE(s) to $\mathcal{F}_{asu}$.

Upon ACCEPT$(\widehat{\texttt{tag}})$ from $\widehat{\texttt{server}}$ in subsession s ($\widehat{\texttt{tag}} \in \widehat{DB}$). If $\widehat{\texttt{tag}}$ is adversarially controlled then send IMPERSONATE$(\widehat{\texttt{tag}})$ to $\mathcal{F}_{asu}$.
Elseif s is the identifier for which $\widehat{\texttt{tag}} = \widehat{\texttt{tag}}_s$ then send ACCEPT(s) to $\mathcal{F}_{asu}$.

Upon $\widehat{\mathcal{A}}$ requesting CORRUPT at $\widehat{\texttt{tag}}_s$. Mark $\widehat{\texttt{tag}}_s$ as corrupted and store its state in $\widehat{DB}$ permanently. In all future executions the output of $\widehat{\texttt{tag}}_s$ is determined by the adversary. Send CORRUPT(s) to $\mathcal{F}_{asu}$.

Fig. 10. Interactions between $\mathcal{F}_{asu}$ and $\widehat{\mathcal{A}}$, $\widehat{\texttt{server}}$, $\widehat{\texttt{tag}}$.

—$\mathcal{S}_{sim}$ prevents the $\widehat{\texttt{server}}$ from outputting $\texttt{accept(tag)}$ in the ideal world when $\mathcal{A}$ tampers with messages created by honest tags in the real world.

Observe that if the RNGs of tags generate true random numbers then the flows of the Flyweight protocol are uniformly distributed and independent. Under this assumption the real and ideal world simulations might differ only when the simulator $\mathcal{S}_{sim}$ intervenes to prevent ACCEPT$(\widehat{\texttt{tag}})$ in the ideal world. For this to happen the messages created by honest tags in the real world must have been tampered by $\mathcal{A}$, so that there is a collision between the (tampered) real world outputs of $\texttt{tag}$ and the idealized outputs of $\widehat{\texttt{tag}}_s$ in a subsession s. Since we assume that RNGs are truly random, the adversary cannot count on this happening with more than negligible probability.

More concretely, this will happen with probability at most $2^{1-n}mL$, where n is the length of the random numbers generated, m is the number of tags managed by the Server, and L is the total number of tag interrogations. This is negligible in the *security parameter* n if we assume that m and L are polynomially bounded in n. It follows that if $\mathcal{Z}$ can distinguish real simulations with pseudo RNGs from ideal simulations, then it can also distinguish real simulations with pseudo RNGs from real simulations with true RNGs. This will lead to a contradiction, if the numbers generated by a pseudo RNG are indistinguishable from random by a PPT adversary. □

10.1 A concrete security reduction, I

A security reduction must relate distinguishing real-vs-ideal worlds to distinguishing pseudo-vs-true randomness. To accomplish this, faithfully simulate the real world and use $\mathcal{Z}$ as a distinguisher. For a true RNG, we get the ideal simulation subject to collisions, for which the probability is at most $2^{1-n}mL$. If we also take into account the advantage $Adv_{RNG}(q,t)$ of distinguishing a pseudo RNG from a true RNG, in

Simulated interactions by $\mathcal{S}_{sim}$, II

Upon request $\texttt{init}(\widehat{server})$ from $\widehat{\mathcal{A}}$. $\widehat{\texttt{server}}$ sends *query* to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, \widehat{\texttt{tag}})$ from $\widehat{\mathcal{A}}$. If no numbers in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ have been marked then $\widehat{\texttt{tag}}_s$ marks r_1 and sends r_1 to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, r_1, \widehat{\texttt{server}})$ from $\widehat{\mathcal{A}}$. If r_1 is the first unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ then $\widehat{\texttt{server}}$ marks r_1, r_2 in $flow(s)$ and sends r_2 to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, r_2, \widehat{\texttt{tag}})$ from $\widehat{\mathcal{A}}$. If r_2 is the first unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ then:

If $|session\_id(\texttt{tag}_s)| = 1$ then $\widehat{\texttt{tag}}_s$ marks r_2, r_3, outputs $\texttt{update}(s)$ and sends $r = r_3$ to $\widehat{\mathcal{A}}$.
Else $\widehat{\texttt{tag}}$ marks r_2, r_4 and sends $r = r_4$ to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, r, \widehat{\texttt{server}})$ from $\widehat{\mathcal{A}}$. .

If $r = r_3$ is the first unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ and $|session\_id(\texttt{tag}_s)| = 1$ then $\widehat{\texttt{server}}$ marks all remaining values in $flow(s)$, outputs $\texttt{update}(s)$ and $\texttt{accept}(\widehat{\texttt{tag}}_s)$.
Elseif $r = r_4$ is the first unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ then $\widehat{\texttt{server}}$ marks r_3, r_4 and sends $r = r_3$ to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, r_3, \widehat{\texttt{tag}})$ from $\widehat{\mathcal{A}}$. If r_3 is the first unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ and $|session\_id(\texttt{tag}_s)| > 1$ then $\widehat{\texttt{tag}}_s$ marks the remaining numbers and sends r_5 to $\widehat{\mathcal{A}}$.

Upon request $\text{SEND}(s, r_5, \widehat{\texttt{server}})$ from $\widehat{\mathcal{A}}$. If r_5 is the only unmarked number in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ then $\widehat{\texttt{server}}$ marks it and outputs $\texttt{update}(s)$ and $\texttt{accept}(\widehat{\texttt{tag}}_s)$.

Fig. 11. Interactions between $\widehat{\mathcal{A}}$, $\widehat{\texttt{server}}$, and $\widehat{\texttt{tag}}$.

which q is the number of numbers drawn from RNG, and t the computational time (steps) taken to draw a number, then the advantage of distinguishing real from ideal is bounded by, $2^{1-n}mL + Adv_{RNG}(mL, T + mL)$, where T is the combined time complexity of $\mathcal{Z}$ and $\mathcal{A}$.

10.2 Resiliency

The refresh functionality endows the Flyweight RFID protocol with forward and backward security.

THEOREM 10.2. *The Flyweight RFID protocol with* refresh *functionality UC-realizes $\mathcal{F}_{rasu}$ if* refresh *supports resiliency.*

PROOF. We extend the proof of Theorem 10.1 to capture the specifications of $\mathcal{F}_{rasu}$. $\mathcal{S}_{sim}$ simulates copies of parties $\widehat{\mathcal{A}}$, $\widehat{\texttt{server}}$, $\widehat{\texttt{tag}}$, objects $\widehat{DB}$, triggers etc, and faithfully translates real world runs between $\{\mathcal{A}$, Server, tag$\}$ into their ideal world counterparts between $\{\widehat{\mathcal{A}}$, $\widehat{\text{server}}$, $\widehat{\texttt{tag}}_s\}$, adhering to the specifications of $\mathcal{F}_{rasu}$. The refresh functionality requires one additional REFRESH item for the simulations in Figure 10—see Figure 12. Also, $flow(s)$ in Item 2 has one more number: $flow(s) = (r_1, r_2, r'_5, r_3, r_4, r_5)$. For the simulations in Figure 11, to deal with the case when *refresh* is ON, the $\text{SEND}(s, r_1, \widehat{\texttt{server}})$ command, Item 3, Figure 11, needs to be modified and a new $\text{SEND}(s, r, r'_5, \widehat{\texttt{tag}})$ item added—see Figure 13.

Additional simulated interactions, I

Upon REFRESH at $\widehat{\texttt{tag}}_s$ in subsession s. If there is a record *session_id*(tag) that contains s then send REFRESH(s) to $\mathcal{F}_{rasu}$.

Fig. 12. Interactions between $\mathcal{F}_{asu}$ and $\widehat{\mathcal{A}}, \widehat{\texttt{server}}, \widehat{\texttt{tag}}$.

Resilience against forward security attacks in the real world follows from our as-

Additional simulated interactions, II

Upon request SEND$(s, r_1, \widehat{\texttt{server}})$ from $\widehat{\mathcal{A}}$. If there are numbers r_1, r_2 in $flow(s)$ of $\widehat{\texttt{tag}}_s$ in $\widehat{DB}$ and no number has been marked then: If *refresh* is OFF then $\widehat{\texttt{server}}$ marks r_1, r_2 and sends r_2 to $\widehat{\mathcal{A}}$. If *refresh* is ON then $\widehat{\texttt{server}}$ marks r_1, r_2, r_5' and sends (r, r_5') to $\widehat{\mathcal{A}}$.

Upon request SEND$(s, r, r_5', \widehat{\texttt{tag}})$ from $\widehat{\mathcal{A}}$. If r_2 is the first unmarked number in $flow(s)$ then $\widehat{\texttt{tag}}_s$ marks r_2, r_5', r_3, outputs UPDATE, and sends r_3 to $\widehat{\mathcal{A}}$.

Fig. 13. Interactions between $\widehat{\mathcal{A}}$, $\widehat{\texttt{tag}}$.

sumption that *refresh* is resilient (Definition 6.1). In the ideal world linking past flows separated by refreshment is prevented by the functionality $\mathcal{F}_{rasu}$ even if the tag gets compromised (the entries in record *session_id*(tag) are discarded—Item 4, Figure 9).

Resilience against backward security is similar. In the real world it follows from our assumption that *refresh* is resilient (Definition 6.1). In the ideal world linking future flows separated by refreshment is prevented by the functionality $\mathcal{F}_{rasu}$ (again Item 4). Thus, as in Theorem 1, the environment $\mathcal{Z}$ cannot distinguish the real from the ideal simulations. □

10.3 A concrete security reduction, II

Let $Adv_{RNG}(n, q, t, s)$ be a lower bound on the probability of predicting the next n-bit number drawn from a PRNG of a tag, if no more than q numbers are drawn from it, with t, s bounds on the time and space complexity. If we assume that the Server refreshes all tags prior to q numbers being drawn from their PRNGs, then the probability of distinguishing real from ideal world executions is bounded by:

$$2^{1-n} mL + Adv_{RNG}(n, q, t, s).$$

10.4 A resilient refresh function

We give an informal proof that the function

$$refresh(K; R, state) = g_{tag}(K \oplus R \oplus state)$$

proposed in Section 8 supports resiliency. Let $\mathcal{RNG}_m$ be a family of PRNGs (defined by the parameter m) for which the probability that a drawn number can be distinguished from random is no better than p_0, provided no more than M numbers are drawn, given a *history* of numbers drawn earlier from this and other generators.

Let $X_m \in \mathcal{RNG}_m$ and Y_m be the refreshed PRNG. The state of Y_m is randomized, so it is uniform in $\mathcal{RNG}_m$. We claim that one cannot distinguish pairs of numbers drawn from (X_m, Y_m) from random pairs, with probability better than p_0, if no more than M numbers are drawn from X_m, Y_m. Suppose, by contradiction, that one can distinguish $(x, y), (x', y'), \ldots,$ from random with probability better than p_0. Then a Distinguisher can use Y_m (a random generator in $\mathcal{RNG}_m$) as an oracle to distinguish $x, x', \ldots,$ from random with probability better than p_0. This is a contradiction. This also implies unlinkability: if x drawn from X_m can be linked to x' in *history* with probability better than p_0, then it is not random.

11. CONCLUSION

Secret sharing (sharing a key) and threshold cryptography (sharing a cryptographic function) are powerful cryptographic mechanisms that support fault-tolerant multi-party communication and computation. Similarly sharing clocks, even if these are only loosely synchronized, will thwart replay attacks.

In this paper we have shown that by sharing a loosely synchronized stream of pseudo-random numbers we can implement a lightweight authentication mechanism that: (i) guarantees session unlinkability with forward and backward security and (ii) thwarts man-in-the-middle relay attacks, in a strong security framework. Furthermore, for appropriate implementations, we can guarantee that the failure rate is kept below a given threshold through regular refreshing.

REFERENCES

Arapinis, M., Delaune, S., and Kremer, S. 2008. From One Session to Many: Dynamic Tags for Security Protocols. In *Proceedings, 15th Int. Conf. Logic for Prog. Art. Intell. and Reasoning (LPAR'08)*. Lecture Notes in Artificial Intelligence, vol. 5330. Springer, 128–142.

Avoine, G. 2010. http://www.avoine.net/rfid/.

Avoine, G., Buttyán, L., Holczer, T., and Vajda, I. 2007. Group-Based Private Authentication. In *WOWMOM*. IEEE, 1–6.

Avoine, G. and Oechslin, P. 2005. A scalable and provably secure hash-based rfid protocol. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE Computer Society, Washington, DC, USA, 110–114.

Barak, B. and Halevi, S. 2005. A model and architecture for pseudo-random generation with applications to /dev/random. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 203–212.

Bengio, S., Brassard, G., Desmedt, Y., Goutier, C., and Quisquater, J.-J. 1991. Secure implementations of identification systems. *J. Cryptology 4,* 3, 175–183.

Burmester, M. and de Medeiros, B. 2008. The Security of EPC Gen2 Compliant RFID Protocols. In *ACNS*, S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, Eds. Lecture Notes in Computer Science, vol. 5037. Springer, 490–506.

Burmester, M. and de Medeiros, B. 2009. On the Security of Route Discovery in MANETs. *IEEE Transactions on Mobile Computing 8,* 9, 1180–1188.

Burmester, M., de Medeiros, B., and Motta, R. 2008a. Provably Secure Grouping-Proofs for RFID Tags. In *CARDIS*, G. Grimaud and F.-X. Standaert, Eds. Lecture Notes in Computer Science, vol. 5189. Springer, 176–190.

Burmester, M., de Medeiros, B., and Motta, R. 2008b. Robust, Anonymous RFID Authentication with Constant Key-Lookup. In *ASIACCS 2008*, M. Abe and V. D. Gligor, Eds. ACM, 283–291.

Burmester, M., de Medeiros, B., Munilla, J., and Peinado, A. 2009. Secure EPC Gen2 Compliant Radio Frequency Identification. In *ADHOC-NOW*, P. M. Ruiz and J. J. Garcia-Luna-Aceves, Eds. Lecture Notes in Computer Science, vol. 5793. Springer, 227–240.

Burmester, M., de Medeiros, B., Munilla, J., and Peinado., S. 2009. Secure EPC Gen2 Compliant Radio Frequency Identication. Tech. Rep. E-print #2009/147, International Association for Cryptological Research.

Burmester, M., Le, T. V., and de Medeiros, B. 2006. Towards provable security for ubiquitous applications. In *ACISP*, L. M. Batten and R. Safavi-Naini, Eds. Lecture Notes in Computer Science, vol. 4058. Springer, 295–312.

Burmester, M. and Munilla, J. 2009. A Flyweight RFID Authentication Protocol. Tech. Rep. (No proceedings), Workshop on RFID Security 2009, RFIDSec2009, June 30 - July 2, 2009, Leuven, Belgium.

Burmester, M., van Le, T., and de Medeiros, B. 2006. Provably secure ubiquitous systems: Universally composable RFID authentication protocols. In *Proceedings of the 2nd IEEE/CreateNet International Conference on Security and Privacy in Communication Networks (SECURECOMM 2006)*. IEEE Press.

Burmester, M., Van Le, T., De Medeiros, B., and Tsudik, G. 2009. Universally Composable RFID Identification and Authentication Protocols. *ACM Trans. Inf. Syst. Secur. 12,* 4, 1–33.

Chen, C.-L. and Deng, Y.-Y. 2009. Conformation of EPC Class 1 Generation 2 standards RFID system with mutual authentication and privacy protection. *Engineering Applications of Artificial Intelligence, Elsevier, In Press, Corrected Proof.*

Choi, E. Y., Lee, D. H., and Lim, J. I. 2009. Anti-cloning protocol suitable to epcglobal class-1 generation-2 rfid systems. *Comput. Stand. Interfaces 31,* 6, 1124–1130.

Coppersmith, D., Krawczyk, H., and Mansour, Y. 1994. The shrinking generator. In *Proc. Advances in Cryptology (CRYPTO 1993)*. LNCS. Springer, 22–39.

Dimitriou, T. 2006. A secure and efficient RFID protocol that can make big brother obsolete. In *Proc. Intern. Conf. on Pervasive Computing and Communications, (PerCom 2006)*. IEEE Press.

EPC Global EPC tag data standards, vs. 1.3. http://www.epcglobalinc.org/standards/EPCglobal_Tag_Data_Standard_TDS_Version_1.3.pdf.

Eun Young Choi, D. H. L. and Lim, J. I. 2008. Anti-cloning protocol suitable to epcglobal class-1 generation-2 rfid systems. *Computer Standards & Interfaces Available online, In press, Corrected Proof.*

Goldreich, O., Goldwasser, S., and Micali, S. 1986. How to construct pseudorandom functions. *Journal of the ACM 33,* 4.

Henrici, D. and Müller, P. M. 2004. Hash-based enhancement of location privacy for radio-frequency identification devices using varying identifiers. In *Proc. IEEE Intern. Conf. on Pervasive Computing and Communications*. 149–153.

Hu, Y., Perrig, A., and Johnson, D. B. 2006. Wormhole attacks in wireless networks. *IEEE Journal on Selected Areas in Communications 24*, 370–380.

Huang, H.-H. and Ku, C.-Y. 2009. A RFID Grouping Proof Protocol for Medication Safety of Inpatient. *J. Medical Systems 33,* 6, 467–474.

Juels, A. and Weis, S. A. 2009. Defining Strong Privacy for RFID. *ACM Transactions on Information and System Security 13,* 1.

Kelsey, J., Schneier, B., Wagner, D., and Hall, C. 1998. Cryptanalytic attacks on pseudorandom number generators. In *FSE '98: Proceedings of the 5th International Workshop on Fast Software Encryption*. Springer-Verlag, London, UK, 168–188.

Kim, C. H., Avoine, G., Koeune, F., Standaert, F.-X., and Pereira, O. 2008. The Swiss-Knife RFID Distance Bounding Protocol. In *ICISC*, P. J. Lee and J. H. Cheon, Eds. Lecture Notes in Computer Science, vol. 5461. Springer, 98–115.

Lee, H. and Hong, D. 2006. The tag authentication scheme using self-shrinking generator on rfid system. In *Transactions on Engineering, Computing, and Technology*. Vol. 18. 52–57.

Mangard, S., Popp, T., and Oswald, M. E. 2007. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Vol. (ISBN: 0-387-30857-1). Springer.

MICHAHELLES, F., THIESSE, F., SCHMIDT, A., AND WILLIAMS, J. R. 2007. Pervasive RFID and Near Field Communication Technology. *IEEE Pervasive Computing 6,* 3, 94–96.

MOLNAR, D., SOPPERA, A., AND WAGNER, D. 2006. A scalable, delegatable pseudonym protocol enabling ownership transfer of RFID tags. In *Proc. Workshop on Selected Areas in Cryptography (SAC 2005).* LNCS, vol. 3897. Springer.

MUNILLA, J., A.ORTIZ, AND PEINADO, A. 2006. Distance bounding protocols with void-challenges for rfid. In *Internat. Conf. on RFID Security (RFIDSec'06).*

OHKUBO, M., SUZUKI, K., AND KINOSHITA, S. 2003. Cryptographic approach to "privacy-friendly" tags. In *Proc. RFID Privacy Workshop.*

PAISE, R.-I. AND VAUDENAY, S. 2008. Mutual authentication in RFID: security and privacy. In *ASIACCS*, M. Abe and V. D. Gligor, Eds. ACM, 292–299.

PERIS-LOPEZ, P., HERNANDEZ-CASTRO, J. C., ESTEVEZ-TAPIADOR, J. M., AND RIBAGORDA, A. 2009. LAMED - A PRNG for EPC Class-1 Generation-2 RFID specification. *Comput. Stand. Interfaces 31,* 1, 88–97.

QINGLING, C., YIJU, Z., AND YONGHUA, W. 2008. A Minimalist Mutual Authentication Protocol for RFID System and BAN Logic Analysis. *Computing, Communication, Control and Management, ISECS International Colloquium on 2*, 449–453.

SEO, D., BAEK, J., AND CHO, D. 2005. Secure RFID Authentication Scheme for EPC Class Gen2. In *Proc. 3rd Int. Conf. on Ubiquitous Information Management and Communication (ICUIMC-2009).* 221–227.

SHARMA, S. E., WEISS, S. A., AND ENGELS, D. W. 2003. RFID systems and security and privacy implications. In *Proc. of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 20002).* LNCS, vol. 2523. Springer, 454–469.

SUN, H.-M. AND TING, W.-C. 2009. A Gen2-Based RFID Authentication Protocol for Security and Privacy. *IEEE Transactions on Mobile Computing 99,* 1.

VAN LE, T., BURMESTER, M., AND DE MEDEIROS, B. 2007. Universally Composable and Forward-Secure RFID Authentication and Authenticated Key Exchange. In *Proc. of the ACM Symp. on Information, Computer, and Communications Security (ASIACCS 2007).* ACM Press, 242–252.

VAUDENAY, S. 2007. On Privacy Models for RFID. In *ASIACRYPT*, M. Abe and V. D. Gligor, Eds. ACM, 68–87.

ZHANG, D., MANOLOPOULOS, Y., THEODORIDIS, Y., AND TSOTRAS, V. 2009. *Power Analysis Attacks - Revealing the Secrets of Smart Cards.* Vol. (Encyclopedia of Database Systems, L. Liu and M. Tamer Őzsu (editors)). Springer.