PSP: Private and Secure Payment with RFID

Erik-Oliver Blass[1,a,*], Anil Kurmus[b], Refik Molva[c], Thorsten Strufe[d]

[a] *Northeastern University, Boston, USA*
[b] *IBM Research – Zurich, Switzerland*
[c] *EURECOM, Sophia Antipolis, France*
[d] *Universität Darmstadt, Darmstadt, Germany*

Abstract

RFID can be used for a variety of applications, e.g., to conveniently pay for public transportation. However, achieving security and privacy of payment is challenging due to the extreme resource restrictions of RFID tags. In this paper, we propose PSP – a secure, RFID-based protocol for privacy-preserving payment that supports multiple different payees. Similar to traditional electronic cash, the user of a tag can pay for a service using his tag and so called *coins* of a virtual currency. With PSP, tags do not need to store valid coins, but generate them on the fly. Using Bloom filters, readers can verify the validity of generated coins offline. PSP guarantees privacy such that neither payees nor an adversary can reveal the identity of a user or link subsequent payments. PSP is secure against *invention* and *overspending* of coins, and can reveal the identity of users trying to *double spend* coins. Still, PSP is lightweight: it requires only a hash function and few bytes of non-volatile memory on the tag.

Keywords: RFID, Payment, Privacy, Security, Ecash

---

1. Introduction

Radio Frequency Identification (RFID) systems that were originally targeting simple object identification are used more and more for sophisticated applications such as access control and payment. The idea of having several small and cheap devices wirelessly communicate with a reader is appealing for various scenarios. The Oyster Card [1] is a prominent example of a large scale deployment of contactless smartcards, i.e., powerful RFID tags, enabling convenient payment for public transportation services. In such a scenario, people entering a metro or a bus just quickly hold their tags close to a reader device in front of

*Corresponding author. Full address: College for Computer and Information Science, Northeastern University, 360 Huntington Ave., 202 West Village H, 02115 Boston, USA, Tel: +1-617-383-4072, Fax: +1-617-373-3768

Email addresses: `blass@ccs.neu.edu` (Erik-Oliver Blass[1,]), `zrlkur@ch.ibm.com` (Anil Kurmus), `molva@eurecom.fr` (Refik Molva), `strufe@cs.tu-darmstadt.de` (Thorsten Strufe)

[1] Work done while at EURECOM.

a gate to issue the payment for the transport fee. Similar payment and access control schemes are already in widespread use for many other kinds of applications, like payments in a cafeteria, for vending machines or for road toll. In general, a payee is represented by one or more readers, and "payers" pay for "services" through their prepaid card or tag by means of some non-interactive, wireless payment protocol.

Such payment scenarios, however, raise challenging security and privacy issues. First, an intruder or a malicious user trying to issue bogus payments or impersonate legitimate users should be prevented from access to services. Also, malicious payees impersonating legitimate payees should not be able to fake payments for services they do not provide. The second concern is privacy that is often emphasized both as a user requirement and as a regulatory matter. Privacy requires that neither external adversaries nor payees are able to identify or trace users by exploiting the payment system.

To make matters worse, future applications will, for increased convenience, require usage of one single tag to pay multiple different payees. For example, one should be able to pay metro services *and* toll roads *and* the cafeteria with one single tag, instead of using multiple tags. Finally, readers, as in a metro station or in a bus, are often embedded devices that are not permanently connected to a backend system such as a server. So readers must be able to verify *offline* and within a very short period of time, whether a payment is valid or not.

This payment scenario typically calls for an offline, anonymous electronic payment solution such as the ones targeted by myriads of ecash, payment and micro-payment schemes extensively covered in the literature – e.g., see first seminal papers by [2–4], for micropayment see [5, 6], or see [7] for an overview. Yet, due to the inherent requirement for blind signatures, existing solutions for anonymous, offline payment are based on complex asymmetric cryptography. Customizing these solutions for RFID systems therefore implies prohibitive complexity that exceeds the capacity of existing RFID devices. Due to the strong cost and size constraints, RFID tags neither feature complex asymmetric cryptographic primitives nor large amounts of memory, cf., [8–13]. Typically, hash functions are the only cryptographic primitive feasible on the hardware of RFID tags. Alternative approaches based on time-memory trade-offs, i.e., storing a number of precomputed values on the tag like MilliCent [14] or MicroMint [15], are not suitable either, since tags also have very little non-volatile memory. In conclusion, traditional ecash solutions cannot be applied to RFID.

Related work: The Oyster Card has several drawbacks: its security has been broken [16], it uses a physically large, contactless smartcard being more expensive than today's tiny RFID EPC Gen 2 tags [17], and bank and metro are combined, so there cannot be any privacy for users. Finally, Oyster Card does not support multiple different payees, but only the metro. Similar to Oyster Card, large and expensive contactless smartcards are also used in, e.g., Visa's payWave [18] program or MasterCard's PayPass [19]. Besides their high price, problems with these smartcards' security and privacy have already been reported, cf., [20].

Other RFID payment systems depend on using an additional mobile phone

to confirm payment or read out a barcode on the phone's display [21–24] – which is inconvenient.

To the best of our knowledge, there is no *privacy-preserving*, *secure*, and *offline* payment solution *distinguishing* between bank and multiple *different* payees solely using RFID tags for payment.

This paper presents PSP, a protocol for secure, private, and offline payment suited for RFID tags. Following notions of ecash, the tag serves as a rechargeable *electronic wallet*, it is responsible for repeated "storage" of pre-paid coins of a virtual currency, and manages all communication with the reader. The **main idea** of PSP is that the tag does not physically store coins, but it receives some information from the *bank* to generate a limited number of valid coins on the fly. Using Bloom filters, readers can verify the validity of coins received from tags.

PSP meets the security and privacy requirements raised by RFID based payment as follows:

- An adversary cannot arbitrarily *invent* new coins, i.e., introduce coins into the system for which he did not legitimately pay for through the bank. Along the same lines, malicious payees cannot eavesdrop or steal coins originally targeted to pay other payees.
- *Overspending* coins is impossible: an adversary cannot replay coins from his own payments or stolen coins from other people's payments he eavesdropped.
- Readers are offline and only synchronize periodically, e.g., once a day, with the bank. Yet, an adversary still cannot *double spend* coins: he cannot pay with the same coin twice at *different* readers. If he does, his identity will be revealed at the bank. Revealing the identity of adversaries trying to double spend money has $O(1)$ complexity for the bank, but is impossible for readers.
- Users of RFID tags remain *anonymous* and are *untraceable*: the true identity of a user is hidden to all adversaries and payees. Also, neither payees nor any adversary can trace tags or their users on subsequent payments, i.e., link different payments to the same tag.
- PSP is lightweight: besides evaluating a hash function, tags require only a few bytes of non-volatile memory.
- PSP copes with resource-limited readers. Readers are often embedded devices. Their storage and computational resources are, although orders of magnitude higher compared to tags, still restricted. Regarding the number of tags and coins, complexity for verifying a coin is in $O(1)$.

The sequel of the paper is structured as follows: first, an overview of the main idea behind PSP is given in Section 2; Section 3 presents the adversary model assumed in this paper; Section 4 introduces the general setup of the

system, the bank, tags, readers, notion of time etc; Section 5 then presents PSP in detail: the process of preparing the whole system by the bank, the act of (re-)charging a tag, and the actual payment protocol; Section 6 discusses why PSP is secure, i.e., analyzes its security and privacy properties.

2. Overview

PSP supports payment of multiple different payees, e.g., the metro, cafeterias or toll road systems. Each payee offers a service and is represented by its own set of RFID readers. For the sake of clarity, most of PSP's protocol description will focus on the single payee scenario. Still, PSP supports security and privacy with multiple different payees, and we will carefully discuss multiple payee issues when appropriate in the next sections.

Before starting with the precise description of the system setup and all protocol details in sections 4 and 5, the following paragraphs give an *informal overview* of PSP's basic concepts.

Payment in PSP. The idea of payment with PSP is that each user Alice carries an RFID tag T_{Alice}. Before Alice gets access to a service, she swipes T_{Alice} close to a reader associated to this particular service. Reader and T_{Alice} exchange data, using the PSP protocol. In general, the data exchanged is payment information – in PSP, this is some kind of "digital coins" of money. PSP basically achieves reader authentication and the actual payment by means of interleaving a challenge-response scheme with a commitment scheme.

First, T_{Alice} sends a coin which is a commitment to some *value* to the reader. Then, T_{Alice} performs a challenge-response protocol for reader authentication. As tags and readers do not share keys in PSP, T_{Alice} knows a set of precomputed challenges and matching response pairs provided by the bank. Once the reader is successfully authenticated by T_{Alice}, T_{Alice} completes the payment by revealing the committed *value* to the reader. Before giving access to a service, the reader verifies whether the coin received from Alice is valid and whether it has been spent before. To protect against an adversary trying to eavesdrop and steal a coin, T_{Alice} sends not only a valid coin during payment, but an additional "fake" coin. Only readers, but not an adversary, can distinguish valid from fake coins. Coins are reader dependent, such that a malicious reader (of a malicious payee) cannot impersonate as another payee's reader to steal coins.

It is quite important to point out that PSP aims at *non-interactive payments*. Here, the user does not have to actively acknowledge a payment. For increased convenience, the user is not required to push a button or enter a PIN, but just hold his tag close to the reader to conduct the actual payment.

Charging: Storing Money on Tags. As other ecash (micro-) payment schemes, PSP provides a prepaid payment service: before any payment, Alice has to "charge" her tag with digital coins. Therefore, Alice goes to a bank and gives "real" money to the bank. In return, the bank sends back some information to the tag which is stored on the tag's non-volatile memory. Using this information, the tag can later generate digital coins used for payments.

Eventually, a tag is "exhausted", i.e., all the coins Alice has paid for are spent. Alice can go to the bank and (re-)charge her tag or discard the old tag and buy a new one. During charging, the bank also gives the tag a set of so called verification bits that will be used during above mentioned challenge-response authentication.

Reader Preparation. The bank also prepares each payee's readers. During an initial phase, before any charging of tags, before any payment, and only *once*, the bank prepares a Bloom filter for each reader of each payee. This Bloom filter represents all possible valid coins in the system. Consequently, later during payment, a reader can therewith verify, whether a coin received from a tag is valid or not.

Maintenance. Readers are considered to generally be offline during normal, daily operation. Only periodically, e.g., once a day during the night, the readers of all payees conduct maintenance: readers connect to the bank, send in the coins collected over the day, and receive from the bank information to update their Bloom filters. The bank also verifies coins sent by readers. If a coin sent by a reader is successfully verified, the respective payee is reimbursed with "real" money.

Time. To cope with new tags entering the payment system over time, old ones being discarded, and new coins, time is divided into "epochs" with PSP. The typical duration of an epoch is in the order of several days or *one month*. At the beginning of a new epoch, the bank carries out the abovementioned initial preparation of readers once. Also, the bank and all readers discard all information, i.e., the Bloom filters, predating the last epoch. As a result, Alice might not be able to pay with coins which are older than 2 epochs anymore – coins eventually expire.

PSP's Security and Privacy Properties. The information received from the bank allows Alice's tag to only generate as many coins as she has paid for. Any other generated coin is, with high probability, rejected by readers. If Alice tries to spend one valid coin, i.e., a coin she really paid for, twice at the same reader, this is immediately detected and rejected by this reader. If Alice tries to spend a valid coin on the same day at two different readers, this is detected by the bank during maintenance, and the bank identifies Alice as the origin of such malicious behavior. An adversary looking for a "free ride" cannot impersonate a reader and thereby steal coins from tags, as he would have to authenticate himself to a tag. Readers of malicious payees cannot impersonate as readers of legitimate payees and steal coins blaming others. Finally, no one can establish any correlation among coins even when they originate from the same tag. So, neither an eavesdropping adversary nor payees or even the bank can track Alice.

Bloom Filters. The following is a quick introduction to Bloom filters limited to what is necessary for understanding this paper. For more information, refer to [25]. We use Bloom filters, as they are a *space efficient* data structure representing a set of elements. The following operations are supported. 1.) addBF(x, y) adds a new element y to Bloom filter x. 2.) isElement(x, y) outputs *true*, if y has been added to Bloom filter x, $false$ otherwise. Here, isElement(x, y) is prone to *false positives*: it might output *true* with probabil-

ity P, even if y has not been added to x before. False negatives are impossible. For convenience, isNotElement(x, y) outputs the opposite of isElement(x, y). Finally, 3.) GenerateEmptyBloomFilter(s) returns a new empty Bloom filter of a specified size s.

3. Adversary Model

Within the domain of payment scenarios, we identify two categories of adversaries that differ in their capabilities and goals: *free riders* and *malicious payees*.

3.1. Free Riders

The goal of a free rider adversary is to get access to a service for free. More precisely, a free rider tries to get a service with coins he did not pay for or to spend the same coin more than once. A typical example for a free rider is a malicious tag holder. We assume an active adversary with total control over communication: the free rider can not only listen to wireless communication between tags and readers, but also initiate communication with arbitrary tags and readers. The free rider might act like a man-in-the-middle and *rush*, i.e., he can intercept messages, modify or even selectively block them before forwarding them to their destination. The free rider also sees the "outcome" of a payment, i.e., if a payee provides his service after tag and reader have exchanged some messages. A free rider might also compromise tags. He can read-out all the memory of the tag and tamper with the data and logic stored on a tag. As a result, the tag's behavior might not comply with the protocol any more. If he compromises a tag, there is, of course, no way to prevent him from spending all coins of the tag the original owner paid for. In conclusion, the above free rider adversary is equivalent to the one of [26] or the STRONG adversary of [27].

3.2. Malicious Payees

The second kind of adversary PSP protects against are malicious payees. In addition to full control over the network and some compromised tags like free riders, a malicious payee adversary also controls his readers. These readers can therefore be seen as "compromised", thus they might also not behave in a protocol compliant manner. All information stored on these readers is known to the malicious payee. As mentioned above, PSP is non-interactive, so it is not requiring any kind of acknowledgment from Alice to carry out the payment. All non-interactive payment scenarios are obviously prone to malicious payees requesting more coins from T_{Alice} than Alice "expects" them to do. There is no way to protect against this kind of fraudulent behavior by malicious payees, as non-interactiveness is required for increased user convenience and user acceptance. We conjecture that a malicious payee adversary will not try to cheat Alice in the above way: if Alice loses coins, she might, e.g., check a monthly bill provided by the bank similar to credit card bills. If Alice notices bogus payments on her bill with a certain reader/payee, she will become suspicious

and draw attention to this payee. In this paper, to not draw any unwanted attention, the goal of a malicious payee is to use his readers to steal money from Alice by impersonating as *other* payees' readers, i.e., to receive payments in the name of others or to get more money from the bank than he received coins from legitimate users during legitimate payments.

Note that, without special physical assumptions such as time or distance bounding, Mafia Fraud [28] is generally possible in ecash systems. Preventing Mafia Fraud is out of scope of this paper, but PSP can be extended, e.g., using RFID time bounding protocols, cf., [29, 30].

Free riders and malicious payees are computationally, timewise, and memory-wise bounded to typical "security margins". For example, they cannot invert a hash function.

3.3. Privacy

Typically, users do not want anyone, be it other users, adversaries or even a curious bank, to know anything about payments carried out. More precisely, Alice's true identity should not be revealed during payment, she must remain anonymous. Alice also does not want to be traced by anyone. Subsequent payments done either with the same payee or with different payees should look as if coming from different users to protect against profiling her by linking payments.

In conclusion, users require privacy, unlinkability, and untraceability as formally defined by, e.g., [26] or [31]. We will come back to these definitions in Section 6.3. Note that with Oyster Card, there is no privacy of users [32].

Bank. Preserving users' privacy against a curious bank turns out to be difficult in an RFID-based setup. In classical ecash protocols, users' privacy is protected by blind signature techniques, cf., [2]. However, blind signature techniques involve computationally complex asymmetric cryptography, which is prohibitive on tiny, cheap RFID tags. To still preserve privacy against a curious bank, we therefore assume a threefold bank setup in PSP, where the bank itself is split into three entities, bank0 ("payer accounting office"), bank1 ("payee accounting office"), and bank2 ("escrow office"). It is assumed that there is no cooperation between these entities. Bank0, bank1, and bank2 do obviously not cooperate with free riders or malicious payees.

The complex setup with three bank entities is required, as no entity must be able to link payments to a certain payer and payee, but payees still get reimbursed with real money for valid coins. This results in a property similar to ecash's blind signatures. More details will follow in the next sections. Note that, in the case of a single bank entity, it is still possible to achieve privacy of users at least against readers (see [33]).

For completeness sake, we assume all communication between readers and bank as trusted using traditional security mechanisms. Also, the communication between Alice and the bank, e.g., for charging her tag, uses a secure channel.

4. System Assumptions and Setup

Due to their limited capacity, the most complex operation tags can afford is a cryptographic hash function [8–10, 12, 13, 26, 34, 35]. For convenience, we assume communication between tags and readers to be error-free. As wireless communication is typically prone to static noise, we assume appropriate mechanisms, like CRC and ARQ techniques, to be implemented on lower communication layers. As mentioned in Section 3, the adversary might, however, selectively drop messages. In this case, underlying ARQ mechanisms will give a timeout to PSP.

In sequel to this section, we introduce PSP's components.

4.1. Bank

Besides users and payees, the third stakeholder in a payment system is the bank. If Alice goes to the bank and pays real money, the bank will in return "charge" Alice's tag with coins. Also, the bank will prepare Bloom filters, so enabling readers can verify coin validity. During maintenance, payees' readers will send their coins received during payments to the bank to get real money back. Also during maintenance, the bank will be able to identify adversaries double spending coins.

As discussed in the previous section, the stakeholder bank is split into three entities, a "payer accounting office" (bank0), "payee accounting office" (bank1), and "escrow office" (bank2). While bank1 will deal with readers and payments during maintenance to reimburse coins, it does not have direct access to the name of tag holders. Along these lines, bank0 has access to tag holders' names, but it does not have access to payments. The "escrow office"/bank2 has access to neither readers, nor holders' names. As bank0, bank1, and bank2 are segregated entities, "the bank" cannot reveal which tag performed which payment. This ensures privacy of tags and tag holders even in the absence of expensive blind signatures.

4.2. Money

PSP uses a virtual currency called *coins*. Although we use, by a stretch of language, the notion of *storing* coins on a tag, tags do not directly "store" coins in their non-volatile memory: instead, bank0 will provide tags with information that allows them to *create valid coins* in real-time.

Alice can *charge* a tag with money, i.e., trade in real money ($, £, €, ...) into coins. Alice can buy and (re-)charge her tag only at bank0.

For convenience, we make the following simplifications regarding handling coins, charging, and payment in PSP:

1.) The exchange rate is $1 for one coin. All services Alice can buy will always cost integer multiples of coins, there is no notion of fractions of a coin.

2.) A tag can only be (re-)charged, if all its coins have been spent. Also, a tag can only be charged with $\gamma_{\max}$ coins at a time. Every time Alice wants to charge her tag with $\gamma_{\max}$ coins, bank0 provides Alice's tag with a so called "ID". This ID will be used to generate coins during payments.

As PSP with the above simplifications for better understanding is rather limited, we will extend it and add more flexibility in Section 5.5.

4.3. Per Epoch System Parameters

In PSP, time is divided into consecutive *epochs* $\epsilon_1, \ldots, \epsilon_{256}$. For example, one epoch is one month. Using statistics available over the last recent years with traditional payments, the following averages or expectation values are known. On average *and per epoch* ϵ_i:

There are τ different tags $T_1, \ldots, T_\tau$ in the system.

On average, a tag will spend a total of γ_{avg} coins with all payees during one epoch.

Unused coins will expire after 2 epochs, e.g., after two months, but can later be reimbursed, cf., Section 6.2.

In total, there are $\eta = \tau \cdot \gamma_{\text{avg}}$ coins in the system. Consequently, the number of IDs to generate η coins on average is $\#\text{ID} = \frac{\eta}{\gamma_{\max}}$.

4.4. Payees and Readers

The system consists of a total of ρ readers. Each reader has a unique *Reader ID*, $\text{RID} = 1, \ldots, \rho$. For simplicity, we assume exactly one reader per, so there is no meaningful difference between a reader and the payee it represents. Multiple readers belonging to the same payee can be supported by giving different physical readers the same RID.

Readers are not assumed to be permanently online connected to the bank (bank0, bank1 or bank2) and also cannot exchange data with each other. Instead, readers are offline most of the time and connect to the bank, more precisely bank1, using a certain schedule, e.g., once a day during the night. In conclusion, readers are not synchronized most of the time.

Finally, for cost and reliability reasons, readers are also assumed to be resource restricted, embedded devices. We assume their available storage to be similar to what is available on today's embedded memory technologies, e.g., ≤ 1 GByte.

4.5. Security Parameters

PSP assumes a cryptographic hash function h that can be executed on a tag. Output size of h is 128 bit; we use, e.g., a SHA-1 implementation for RFID tags and truncate the output to 128 bit, cf., [11]. More lightweight hash functions, such as SQUASH for RFID, cf., [35], may alternatively be used. Truncation to 128 bit helps to reduce storage amount on the tag, as we will see in Section 5.7.

PSP uses optimized Bloom filters [25] to store information about all η valid coins during one epoch. We now present the major security properties and parameters required for PSP.

1.) Parameter κ defines the number of different hash functions used for Bloom filters. As h is a cryptographic hash function, we can, instead of using κ *different* hash functions $h_1, \ldots, h_\kappa$, simply define $h_i(x) = h(i, x)$, $1 \leq i \leq \kappa$, where "," denotes an unambiguous pairing of inputs.

2.) Parameter μ defines the storage size of each Bloom filter, i.e., its number of bits. For given κ and η, it is possible to define the size μ of the Bloom filter with $\frac{\mu}{\eta} = \frac{\kappa}{\ln 2}$ such that the probability of any bit in the Bloom filter being set to 1 is $p = \frac{1}{2}$. As a result, the probability of finding a single false positive in the Bloom filter is $P = \frac{1}{2^{\kappa}}$. A false positive in our context is the case where an adversary computes or guesses by chance one single coin which is accidently accepted by the Bloom filter – as described later in higher detail.

3.) Parameter ω is the number of *verification bits* used for reader-to-tag authentication. An adversary is able to impersonate a reader with $2^{-\omega}$ probability to receive one single valid coin from a valid tag. *Note:* If an adversary fails to compute the correct verification bits for one coin from a valid tag, the tag will send him "fake" payments – as described in the following sections in higher detail.

5. Protocol Description

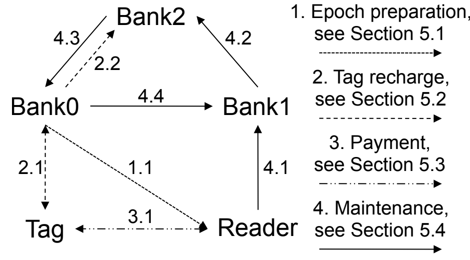

Figure 1: Interactions between tag, reader, and the bank

Before going into PSP details, Figure 1 summarizes party interactions during "epoch preparation", "tag recharge", "payment", and "maintenance" operations. For each operation, Figure 1 depicts the data flow between parties and the sequence of interactions. For example, "2.2" denotes the second interaction of the "tag recharge" operation, and data flows from bank0 to bank2. The following sections will now give details.

5.1. Preparation of new epoch ϵ_{i+1}

For the first epoch, as well as periodically at the end of each epoch (e.g., once a month), bank0 prepares the system for the subsequent epoch ϵ_{i+1} as follows.

1.) Bank0 creates a new 128 bit symmetric *epoch key* $K_{\epsilon_{i+1}}$ and sends it to all payees/readers (Note that this does *not* impose a security problem, see Section 6). Also, bank0 creates two new, empty hash tables called $\Delta_{\epsilon_{i+1}}$ and $\Sigma_{\epsilon_{i+1}}$.

2.) The so called IDs are prepared. In epoch ϵ_{i+1}, $\forall k : 1 \leq k \leq \#\text{ID}$, $\text{ID}_k^{\epsilon_{i+1}}$ can be computed as: $\text{ID}_k^{\epsilon_{i+1}} = h(K_B, k, \epsilon_{i+1})$, where K_B is a 128 bit key only known to bank0 (but *neither* bank1 *nor* bank2).

3.) Bank0 generates Bloom filters for all readers as shown in Algorithm 1. All readers discard the information stored on the second-to-last epoch ϵ_{i-1} (i.e., $\text{BF}^{\epsilon_{i-1}}_{\text{RID}}$ and $\text{spentBF}^{\epsilon_{i-1}}_{\text{RID}}$), before receiving the filters for the following epoch. In summary, a valid *coin* in PSP is a hash commitment to a *precoin*, and

ALGORITHM 1: Preparing Bloom filters

```
for RID := 1 to ρ do
    BF^{ε_{i+1}}_{RID} :=GenerateEmptyBloomFilter(μ);
    for k := 1 to #ID do
        ID^{ε_{i+1}}_k = h(K_B, k, ε_{i+1});
        for c := 1 to γ_max do
            precoin := h(1, RID, ID^{ε_{i+1}}_k, c);
            coin := h(precoin);
            addBF(BF^{ε_{i+1}}_{RID}, coin);
        end
    end
    spentBF^{ε_{i+1}}_{RID} := GenerateEmptyBloomFilter(μ);
    bank0 ⟶ Reader RID : {BF^{ε_{i+1}}_{RID}, spentBF^{ε_{i+1}}_{RID}};
end
```

both can be created simply by knowing a valid RID, a valid ID, and a simple counter c. Value "1" is a publicly known constant, and construction $h(1,\dots)$ (and $h(2,\dots)$, $h(3,\dots)$, ... used later) only serves to avoid ambiguity of hash computations. Section 6.1.1 presents a detailed sample implementation of these constructions.

A valid ID can be generated using key K_B, the epoch ϵ_{i+1}, and a counter k. As K_B is only known to bank0, only bank0 can compute valid IDs. The $\text{BF}^{\epsilon_{i+1}}_{\text{RID}}$ Bloom filter is filled with all possible *coins* that will exist during epoch ϵ_{i+1}. Based on $\text{BF}^{\epsilon_{i+1}}_{\text{RID}}$, reader RID will be in a position to verify whether coins are valid. The $\text{spentBF}^{\epsilon_{i+1}}_{\text{RID}}$ Bloom filter, while empty at the beginning, will later store all coins spent during epoch ϵ_{i+1}. It will enable reader RID to check whether a coin has already been spent, cf., Section 5.3. Note that with one valid ID, up to $\gamma_{\max}$ valid coins for reader RID can be created.

5.2. (Re-)Charging a Tag

In epoch ϵ, Alice wants to buy a new tag or "recharge" an old one. Alice's tag is T_{Alice}. The idea behind charging T_{Alice} is that bank0 gives IDs to T_{Alice} which will enable T_{Alice} to later generate valid coins. To that effect, bank0 maintains a counter ξ^ϵ to store the information about which IDs have already been sold to tags. So, $1 \leq \xi^\epsilon \leq \#\text{ID}$.

1.) Alice gives bank0 the equivalent amount of money to buy $\gamma_{\max}$ coins.

2.) Bank0 computes the yet unused $\text{ID}^\epsilon_{\xi^\epsilon} = h(K_B, \xi^\epsilon, \epsilon)$ and sends the tuple $(\text{ID}^\epsilon_{\xi^\epsilon}, \epsilon)$ to T_{Alice} which stores it in non-volatile memory.

3.) For each coin sold to Alice, bank0 computes so called *verification bits* ν_i, $|\nu_i| = \omega$ bit, and sends them to Alice using Algorithm 2. T_{Alice} stores ν_i. K_ϵ

is the current epoch key (as described in Section 5.1).
Here, $\lfloor h(x) \rfloor_\omega$ is a *truncated* hash value, the first ω bits of output of $h(x)$.

ALGORITHM 2: Computation of verification bits

```
ID^ε_{ξ^ε} := h(K_B, ξ^ε, ε);
for i := 1 to γ_max do
    chall := h(2, ID^ε_{ξ^ε}, i); // Challenge
    ν_i := ⌊h(K_ε, chall)⌋_ω; // Response
    Bank ⟶ T_Alice: {ν_i} ;
end
```

As only the readers and the bank know K_ϵ, Alice can later use *chall* as a challenge to any reader and verify a reader's response using the verification bits ν_i – therewith providing reader authentication. Together with tuple $(\mathrm{ID}^\epsilon_{\xi^\epsilon}, \epsilon)$, T_{Alice} stores a local counter c in its non-volatile memory. c keeps track of how many coins have already been spent by T_{Alice}. Also, a maximum value max is stored, representing the maximum number of coins that can be spent with $\mathrm{ID}^\epsilon_{\xi^\epsilon}$, $\max = \gamma_{\max}$.

4.) The bank adds information to hash tables Δ_ϵ and Σ_ϵ to protect against double spending as described in Algorithm 3. Finally, bank0 sends Δ_ϵ to bank2.

ALGORITHM 3: Bank0 prepares against double spending

```
for RID := 1 to ρ do
    for c := 1 to γ_max do
        prechall := {ξ^ε, ε, c};
        trace := {ID^ε_k, c};
        precoin := h(1, RID, trace);
        coin := h(precoin);
        addHash(Δ_ε, coin, prechall);
        spent := 0;
        identity := {name, spent};
        addHash(Σ_ε, trace, identity);
    end
end
bank0 ⟶ bank2: {Δ_ε};
ξ^ε := ξ^ε + 1;
```

In Algorithm 3, addHash(x, y, z) stores value z in hash table x at key y. In *name*, bank0 stores an (unique) identifier of Alice, e.g., her name or her account number. With *name*, the bank should be able to identify Alice at a later point in time. *spent* is one bit storing the information, whether the user identified by *name* has already spent a coin based on *prechall* at any reader.

So, what basically happens is that bank0 creates the hash table Σ_ϵ to bind each *coin* in the epoch to a different *name*. This will later allow a reverse lookup of which *coin* was spent by which *name*. More precisely, with Δ_ϵ each *coin*

is bound to the information that this *coin* is generated from, i.e., a *prechall* consisting of $\{\xi, \epsilon, c\}$. With Σ_ϵ, this information, represented as hash value *trace*, is then bound to the *name*.

In conclusion, after Algorithm 3, the escrow office (bank2) knows the mapping between *coins* and *prechalls*, and the payer accounting office (bank1) knows the mapping between *prechalls* and *names*. This will be the basis for periodic maintenance later in Section 5.4.

5.3. Payment

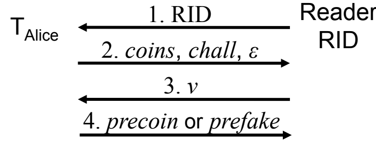

Figure 2: Message flow in PSP

For the sake of simplicity, the price for any service offered by any payee is fixed at 1 coin. This section describes the payment protocol (Figure 2) through which T_{Alice} pays 1 coin to the payee running reader RID. Let the current epoch be ϵ, the current ID used by T_{Alice} to generate coins be ID_ξ^ϵ, the verification bits be ν_i.

Payment Overview: Reader RID initiates the protocol by sending RID to tag T_{Alice}. T_{Alice} responds by sending two *coins*, a valid coin and a fake one. Sending these coins serves both the purpose of confusing a potential adversary and achieving T_{Alice}'s commitment for the payment. T_{Alice} also sends a challenge *chall* and the epoch ϵ of the valid coin. Upon receipt of message 2 and successful verification of the valid coin, the reader replies by sending verification bits v. Finally, if v matches *chall*, the reader is authenticated and T_{Alice} reveals the preimage of the committed valid coin, otherwise the authentication of the reader has failed, and T_{Alice} replies with the preimage of the committed fake coin.

Payment Details: Algorithm 4 presents a more detailed sketch of the payment protocol. In ①, reader RID periodically broadcasts its ID. Using $\text{ID}_{\xi^\epsilon}^\epsilon$, c, and RID, T_{Alice} computes a challenge *chall*, a valid *coin* out of *precoin*, and a (pseudo-)random invalid $fake$ coin out of $prefake$. In ②, T_{Alice} sends *coin*, $fake$, *chall*, and ϵ to the reader. Here, the order of sending *coin* and $fake$ swaps depending on a random bit b: $\text{flip}(b, y, z)$ is $\{y, z\}$ **iff** $b = 0$, and $\{z, y\}$ otherwise. So, T_{Alice} randomly chooses the order of sending valid *coin* and $fake$ coin to mislead the adversary. The adversary cannot distinguish between the valid and the fake coin. Sending *coin* and $fake$ serves as a commitment, where the preimages will be revealed later. The reader verifies, whether one of the two received coins, $coin_1$, $coin_2$, is valid, i.e., is in its Bloom filter $\text{BF}_{\text{RID}}^\epsilon$. Thereby, the reader identifies which coin is the valid coin (called *vcoin* thereafter), see ③. Also, the reader verifies whether this coin has not been spent on any reader before the last scheduled maintenance (check $\text{spentBF}_{\text{RID}}^\epsilon$), and whether the coin has not been spent on this reader since the last maintenance. For the latter, the reader

ALGORITHM 4: Payment procedure

T_{Alice}	Reader RID
`// Receive:` RID	$T_{\text{Alice}} \longleftarrow$ RID : {RID} ①
$chall := h(2, \text{ID}^{\epsilon}_{\xi^{\epsilon}}, c)$;	
$precoin := h(1, \text{RID}, \text{ID}^{\epsilon}_{\xi^{\epsilon}}, c)$;	
$coin := h(precoin)$;	
$prefake := h(3, \text{ID}^{\epsilon}_{\xi^{\epsilon}}, c)$;	
$fake := h(prefake)$;	
$b := \lfloor h(4, \text{ID}^{\epsilon}_{\xi^{\epsilon}}, c) \rfloor_1$;	
$coins := \text{flip}(b, \{coin, fake\})$;	
$c := c + 1$;	
② $T_{\text{Alice}} \longrightarrow \text{RID} : \{coins, chall, \epsilon\}$	`// Receive:` $\{coin_1, coin_2\}$
	if isElement($\text{BF}^{\epsilon}_{\text{RID}}, coin_1$) **then** ③
	$vcoin := coin_1$;
	elseif isElement($\text{BF}^{\epsilon}_{\text{RID}}, coin_2$) **then**
	$vcoin := coin_2$;
	else sleep; **exit**;
	if isNotElement($\text{spentBF}^{\epsilon}_{\text{RID}}, vcoin$) **and** isNotElement($\text{spentList}^{\epsilon}_{\text{RID}}, vcoin$) **then** ④
	addList($\text{spentList}^{\epsilon}_{\text{RID}}, vcoin$);
	$v := \lfloor h(K_{\epsilon}, chall) \rfloor_{\omega}$;
`// Receive:` v	$T_{\text{Alice}} \longleftarrow$ RID : $\{v\}$
`// Authenticate reader` ⑤	
if $v = \nu_c$ **then** `// Finish payment`	
$T_{\text{Alice}} \longrightarrow$ RID : $\{precoin\}$	`//`*received* `is` *precoin* `or` *prefake*
else $T_{\text{Alice}} \longrightarrow$ RID : $\{prefake\}$	**if** $h(received) = vcoin$ ⑥ **then** openBarrier;
	else
	addList($\text{reimburseList}^{\epsilon}_{\text{RID}}, vcoin$);
	sleep; **exit**;
	end
	end
	else sleep; **exit**;

maintains a simple list, $\text{spentList}^{\epsilon}_{\text{RID}}$. If *vcoin* passes the above tests, see ④, *vcoin* is added to $\text{spentList}^{\epsilon}_{\text{RID}}$, the reader computes the truncated hash-value of *chall* using K_{ϵ}, and sends the result back to T_{Alice}. T_{Alice} verifies received verification bits to authenticate the reader. In ⑤, if the verification bits match, T_{Alice} sends *precoin*, the preimage of the *coin* to the reader. Otherwise, T_{Alice} assumes malicious behavior and sends *prefake*, the preimage of *fake*. This exchange of *precoin* or *prefake* after the two coins achieves the commitment of T_{Alice} without allowing an adversary (impersonating a legitimate reader) to determine if the coin it receives is a fake. In ⑥, the reader verifies if the hash of the received preimage (*precoin* or *prefake*) matches coin. In case the hash does not match, *vcoin* is included in another simple list, $\text{reimburseList}^{\epsilon}_{\text{RID}}$. Also,

if the protocol gets somehow interrupted, and the reader does not receive the last message, the reader will include *vcoin* in $\text{reimburseList}^{\epsilon}_{\text{RID}}$. Although the requested services will not be provided by the payee in the last two cases, and the coin is "spent", coins on $\text{reimburseList}^{\epsilon}_{\text{RID}}$ can be later reimbursed to Alice by the bank as described in Section 6.2. In general, if the reader suspects any misbehavior or cheating during PSP, it sleeps (**sleep**) for a reasonable amount of time, e.g., 10 seconds, and exits protocol execution (**exit**). Also, an alarm might go off, security personnel arrives etc.

5.4. Periodic Maintenance

Each reader connects to the "payee accounting office", i.e., bank1. Readers might connect simultaneously, they may select to connect at arbitrary times or using some kind of load balancing mechanism. The current epoch is ϵ_i.

1.) All readers send their $\{\text{spentList}^{\epsilon_{i-1}}_{\text{RID}}, \text{spentList}^{\epsilon_i}_{\text{RID}}, \text{reimburseList}^{\epsilon_{i-1}}_{\text{RID}}, \text{reimburseList}^{\epsilon_i}_{\text{RID}}\}$, with $1 \leq \text{RID} \leq \rho$, to bank1. All readers remove all entries from their spentLists and reimburseLists.

2.) Jointly, bank0, bank1, and bank2 check the lists, whether the included entries, *coin*s of the form $h(h(1, \text{RID}, \text{ID}^{\epsilon}_{\xi^{\epsilon}}, c))$, have already been spent. Algorithm 5 shows this maintenance operation for bank0, bank1, and bank2 separately.

Bank1 starts by forwarding a spentList entry to bank2. Upon receipt, bank2 uses Δ_{ϵ} to do a reverse lookup to get $prechall = \{\xi^{\epsilon}, \epsilon, c\}$, i.e., the information necessary to compute the *trace* (corresponding to an ID and counter pair). As the *trace* can only be computed by bank0, bank2 sends *prechall* to bank0.

Upon receipt, bank0 computes the *trace* and can therewith check whether a coin based on this *trace* has been spent already – using Σ_{ϵ}. If such a coin has already been spent, bank0 can take appropriate countermeasures against that particular double spending user with identity *name*. Finally, bank0 computes all possible coins based on *trace* for each reader and stores $\{\text{RID}, coin\}$ pairs in a set $\mathbb{C}$. This set $\mathbb{C}$ is sent to bank1.

Upon receipt of $\mathbb{C}$, bank1 first checks whether the original *entry* is on a reimburseList of reader RID and not on *any* other reader's spentList. If so, bank1 will ask reimburse this coin. Details have been omitted from Algorithm 5, as this again requires sending *entry* to bank2 and bank2 sending *prechall* to bank0. Bank0 can then reimburse *name*. Finally, bank1 checks whether the original *entry* was indeed paid for reader RID. If so, bank1 sends real money to reader RID, otherwise it reports this reader as malicious. Finally, bank1 forwards coins in $\mathbb{C}$ to each individual reader. Each reader adds such a coin to its spentBF Bloom filters: $\text{addBF}(\text{spentBF}^{\epsilon}_{\text{RID}}, coin')$. So in conclusion, after maintenance, all readers are synchronized: each reader has all coins spent so far stored in its spentBF filter.

Therewith, also the segregation of bank0, bank1, and bank2 becomes clear: bank2 removes the RID information of the coin, so neither bank0 nor bank1 can link a specific payment of a payer to a payee.

ALGORITHM 5: Periodic maintenance

//bank1
for $\epsilon \in \{\epsilon_{i-1}, \epsilon_i\}$ **do**
  for RID := 1 **to** ρ **do**
    foreach $entry \in \text{spentList}^{\epsilon}_{\text{RID}}$ **do**
      bank1 → bank2 : $\{entry\}$;
      //upon receipt of set $\mathbb{C}$ of
      //pairs $\{\text{RID}', coin'\}$ from bank0
      if $entry \in \text{reimburseList}^{\epsilon}_{\text{RID}}$ **and** $entry \notin \text{spentList}^{\{\epsilon_{i-1},\epsilon_i\}}_{\neq\text{RID}}$ **then**
        reimburseCoin($entry$);
      end
      if $\{\text{RID}, entry\} \in \mathbb{C}$ **then** sendMoney(RID);
      else reportReader(RID);
      foreach $\{\text{RID}', coin'\} \in \mathbb{C}$ **do**
        bank1 ⟶ Reader RID′: $\{coin'\}$;
      end
    end
  end
end

//bank2,
//upon receipt of *entry* from bank1
$\{\xi^{\epsilon}, \epsilon, c\} := \text{getValue}(\Delta_{\epsilon}, entry)$; // *prechall*
bank2 → bank0 : $\{\xi^{\epsilon}, \epsilon, c\}$;

//bank0
//upon receipt of $\{\xi^{\epsilon}, \epsilon, c\}$ from bank2
$trace := \{h(K_B, \xi^{\epsilon}, \epsilon), c\}$;
$\{name, spent\} := \text{getValue}(\Sigma_{\epsilon}, trace)$;
if *spent=1* **then** reportIdentity($name$);
else
  modifyHashValue(Σ_{ϵ}, $trace$, $\{name, 1\}$);
  $\mathbb{C} := \emptyset$;
  for $j := 1$ **to** ρ **do**
    $precoin := h(1, j, trace)$;
    $coin := h(precoin)$;
    $\mathbb{C} := \mathbb{C} \cup \{\text{RID}, coin\}$;
  end
  bank0 → bank1 : $\{\mathbb{C}\}$;
end

With getValue(x, y), hash table x is queried with key y, and the corresponding value is returned. ModifyHashValue(x, y, z) sets the value belonging to key y in hash table x to z.

5.5. Adding Flexibility

In this section, we extend PSP with respect to more flexibility of charging tags and payment as well as distributing the bank0/1/2's workload to "proxies".

Money. Instead of charging tags only with $\gamma_{\max}$ coins all at once, tags can be charged with sets of coins called *packs* π. Possible packs are, for example, $\pi \in \{10, 20, 50\}$ coins. So, Alice can buy packs of 10 coins, 20 coins, 50 coins, as well as combinations thereof. For every pack of coins that Alice buys at bank0, bank0 will handout one ID to Alice. Still, a tag can never be charged with more than a total of $\gamma_{\max}$ coins on a tag simultaneously. Also, Alice cannot buy more than, say, up to 3 packs per charge. If a tag stores coins from more than one pack, it will always completely deplete one pack for payment before using coins generated out of another pack. (Re-)Charging is only allowed, as soon as the total number of coins on the tag is less or equal than 9 coins. (This could be mitigated by forcing bank0 to "buy back" the remaining 9 coins, i.e., reimbursing Alice using the technique of Section 6.2.) If a tag is recharged, it will first deplete the remaining old (≤ 9) coins before using the new ones. In conclusion, simultaneously, a tag has up to $\gamma_{\max}$ coins stored in a total of ≤ 4 packs.

Again utilizing statistics, it is known on average how many packs $\pi_i \in \{10, 20, 50\}$ of coins will be bought by users (including a safety margin) during one epoch. The expected number of packs of size 10, $\pi_i = 10$, is η_{10}, η_{20} for packs $\pi_i = 20$, and η_{50} for $\pi_i = 50$. In conclusion, $10 \cdot \eta_{10} + 20 \cdot \eta_{20} + 50 \cdot \eta_{50} = \eta$.

Now, IDs can be computed as: $1 \leq i \leq 3$, $\pi_i \in \{10, 20, 50\}$, $\forall k : 1 \leq k \leq \eta_{\pi_i}$, $\mathrm{ID}_k^{\pi_i,\epsilon} = h(K_B, i, k, \epsilon)$. Note that *prechall* now has to include i, i.e., *prechall* $:= \{\xi^\epsilon, \epsilon, c, i\}$. Bank0 maintains 3 *counters* $(\xi_{10}^\epsilon, \xi_{20}^\epsilon, \xi_{50}^\epsilon)$ to store the information about which IDs have already been sold to tags. So, $1 \leq \xi_{10}^\epsilon \leq \eta_{10}$, $1 \leq \xi_{20}^\epsilon \leq \eta_{20}$, $1 \leq \xi_{50}^\epsilon \leq \eta_{50}$. During charging, Alice gives the equivalent amount of money to buy $i \leq 3$ packs of coins, $\pi_i \in \{10, 20, 50\}$. In return, Alice's tag T_{Alice} receives and stores i tuples consisting of $\mathrm{ID}_{\xi_{\pi_i}^\epsilon}^{\pi_i,\epsilon}$, counter $c_i := 1$, $max_i := \pi_i$, and verification bits $\nu_{i,j}$, $1 \leq j \leq \pi_i$.

If a payee's service costs a total of α coins instead of 1 coin, reader RID broadcasts α together with RID. As soon as T_{Alice} receives this broadcast, it executes the payment protocol of Algorithm 4 α times.

Proxies. To decrease the workload of bank0/1/2, we introduce *proxies*. Readers are not directly, i.e., physically, connected to bank1, but multiple readers are grouped together and physically connected to one of bank1's *proxy* devices, more powerful computers. For example, readers of metro stations in close physical distance are physically connected to one proxy, multiple readers at one toll station connect to one proxy in the evening. A proxy is connected physically to bank1 and will carry out all communication between bank1 and readers. Readers, even the ones connected to the same proxy, are not assumed to be permanently connected to their proxies. This would be impossible in many situations, e.g., for readers in public buses. Readers cannot exchange data with each other and are offline most of the time. Readers connect to their proxies once a day for maintenance. Proxies could collect all readers' spentLists and

reimburseLists and relay them to bank1. Proxies do not need to be permanently connected to bank1, but only once a day. As with bank0, bank1, and bank2, we assume the proxies, but not the readers, are trusted based on the features of some tamper-proof hardware. During maintenance, bank1 does not compute and send spent coins to all readers, but only sends $trace := \{\mathrm{ID}^{\epsilon}_{\xi^{\epsilon}}, c\}$, as in Algorithm 5, to all proxies. Proxies then compute and send coins for all their attached readers, $coin := h(h(1, \mathrm{RID}, \mathrm{ID}^{\epsilon}_{\xi^{\epsilon}}, c))$. For analytical purposes, we subsequently consider σ proxies in the system.

5.6. Real World Parameters

To get an idea about its general feasiblity, we evaluate PSP using "real world" parameters from an ecash-like scheme currently in use: the Oyster Card. Due to the lack of availability of real world figures for a *multi-payee* payment system, we have to assume system parameters similar to the *single payee* Oyster Card. We claim this to be a fair assumption, as the number of payees does not have any impact on neither tags' or readers' computational effort, nor on the storage requirements. As PSP is a prepaid scheme, only the total number of coins ($\gamma_{\max}$) that can be spent in between two charging operations influences PSP's storage costs. We will discuss multi-payee complexity implications for bank0, bank1, and bank2, though.

Between 2003 and 2007, 10^7 Oyster Cards have been issued [36] from which $\tau = 5 \cdot 10^6$ are in use at the same time [32] (some Oyster Cards are not rechargeable and dropped after use). In 2007, the total revenue of public transport with buses and metro in London was 2.420 billion GBP [37]. If all transport would have been paid with Oyster Cards (still traditional methods of payment, such as cash, are used), then $\eta \approx 10^8$ for each epoch of a month. Adopting the Oyster Card setting, we assume $\gamma_{\max} = 80$.

In London, there are 270 metro stations and 6,800 buses [38, 39]. We assume that there are on average 50 readers per metro station and one reader per bus. As a total, we assume $\rho = 20,000$ readers. By assuming $\sigma = 20$ proxies in the system, each proxy is on average associated to 1,000 readers.

We choose $\kappa = 22$, resulting in 2^{-22} probability of a single false positive. With $\omega = 1$, the adversary can impersonate a reader with 50%, but as discussed in Section 6, this is acceptable in the overall scenario.

Finally, $|name| = 32$ bit should be sufficient to uniquely identify a single account or user of the payment system.

5.7. Space Analysis

Based on the more flexible extension of PSP and the real world parameters, we can do the following evaluation:

Tag. T_{Alice} stores 4 IDs, 4 c_i, 4 ϵ_i, 4 $\max_i$ to be able to generate coins. $|\mathrm{ID}| = 128$ bit, $|\max_i| = |c_i| = 6$ bit, $|\epsilon_i| = 8$ bit. This sums up to 592 bit. Also, T_{Alice} stores $(\gamma_{\max} \cdot \omega)$ verification bits ν, i.e., $80 \cdot 1 = 80$ bit. Hence, each tag needs to store 672 bit = 84 byte in its non-volatile memory. This is feasible, e.g., with Alien Technology's prominent Higgs-3 RFID-tag [17], featuring 800 bit of

non-volatile storage. Compared to the Oyster Card, featuring 1 KByte (=8192 bit) storage [40], this is much less and thus leads to cheaper tags in terms of production costs.

Generally for tags, computational complexity is important. However, computational complexity is low with PSP: for payment, the tag has to do only hash evaluations, simple arithmetic, and to wirelessly send $3 \cdot 128 + 8 = 392$ bit to the reader. Based on related work [8–10, 12, 13, 26], we consider that these operations are feasible on tags.

Reader. A standard Bloom filter with false-positive probability P and η elements of a set to represent requires $\mu = \frac{\log_2(\frac{1}{P}) \cdot \eta}{\ln 2}$ bit of storage [25]. So with $\eta = 10^8$, each reader needs $\mu \approx 378$ MByte of storage for the BF Bloom filter and the same for the spentBF Bloom filter. As BF and spentBF are required for the current and second-to-last epoch, this would amount to a total of $4 \cdot 378 \approx 1.5$ GByte per reader. However, standard Bloom filters are not space optimal. Optimizations of Bloom filters can achieve lower storage requirements, such as $\log_2(\frac{1}{P})$ bit per element represented in the filter. For example, [41] suggest to build the Bloom filter with a single hash function instead of $\kappa \gg 1$ hash functions. In that optimized version, the hash outputs $h_1(x)$ of all coins are Golomb encoded and partitioned into blocks. The isElement function that looks for *coin* is implemented by selectively decoding one block and looking for the hash output $h_1(coin)$. The data structure of [41] maintains exactly the same properties as a standard Bloom filter, i.e., false-positive probability P, but requires only $\log_2(\frac{1}{P})$ bit storage per element. In our case, this optimization would results in $\approx$ 262 MByte total storage for each BF filter. Similar to Counting Bloom Filters, [41] allows furthermore for deleting coins out of the data structure, superseding the spentBF Bloom filters on readers. The latter helps in avoiding false positives while checking whether a coin has been spent. Algorithm 4 can therefore be changed such that a reader accepts a coin if it is in its BF Bloom filter and at the same time not on its current spentList. Instead of $4 \cdot 378 \approx 1.5$ GByte, this would keep storage close to a total of $2 \cdot 262 = 524$ MByte for both epochs. Finally, per day, each reader needs on average to store $\frac{10^8}{30 \cdot 20{,}000} \approx 170$ coins, i.e., ≈ 3 KByte on both spentLists for the two epochs. Readers at frequently used metro stations will require more memory for spentLists, but this will still be in the order of magnitude of KBytes per day. Also, a small amount of memory is required for the two reimburseLists. In conclusion, the total amount of memory is considerably less than 1 GByte which should be feasible even on restricted reader hardware.

Computational complexity for the reader is also low. The isElement function required for validating a coin is cheap: it consists of performing an efficient, low complexity ($O(1)$) Golomb-decoding of a fixed length block and searching for $h_1(coin)$ therein [41].

Bank. The bank, i.e., bank0, bank1, and bank2, needs to store Δ and Σ for the current and last epoch. In Δ, for all 10^8 coins and all $20,000$ readers a *prechall* has to be stored. With $|prechall| = |\xi| + |\epsilon| + |c| + |i|$, $|i| = \log_2 3 \approx 2$ bit, $|c| = \log_2 50 \approx 6$ bit, $|\epsilon = 8$ bit, and $\xi = \log_2 \frac{10^8}{10} \approx 24$ bit (worst case: all packs

bought are 10 coin packs), $|prechall| = 40$ bit worst case. So in conclusion, the two Δ for the two epochs require a total of $2 \cdot 20,000 \cdot 10^8 \cdot 40$ bit ≈ 18 TByte. For hash table Σ, there are a total of $\eta = 10^8$ different $trace = \{\mathrm{ID}^{\epsilon}_{\xi^{\epsilon}}, c\}$ keys, and Σ stores for each key *name* ($|name| = 32$ bit) and *spent* ($|spent| = 1$ bit). For the two epochs, this requires $2 \cdot 10^8 \cdot 33$ bit ≈ 787 MByte storage. While the total resulting ≈ 18 TByte for the two epochs is certainly a huge amount of storage, this is still affordable for a bank: using Amazon's S3 pricing [42], we can roughly estimate storage and access costs per month. Storing on average 9 TByte per month, transferring on average 9 TByte in and out, respectively, as well as $\eta = 10^8$ read and write requests, respectively, sums up to $\approx 4,000$ US$ per month. Compared to the 2.420 billion GBP$\approx$ 3.9 billion US$ of total revenue per year [37] for this single payee, we claim this cost to be acceptable. Note that outsourcing data to Amazon introduces new security and privacy problems which we cannot address. The above computation is only to get an idea of and to justify the storage costs.

Assuming **multiple-payees** instead of only one, linearly influences the storage requirement for the bank. For example, supporting twice as many coins due to twice as much total revenue implies twice as much storage – but probably less than twice as much storage and access costs. In any way, we conjecture that linear complexity and scalability with respect to in storage costs will be acceptable for the bank. Note that the tag's storage requirements do not change with respect to the number of payees.

Computational complexity for the bank is low, too: preparation of a new epoch has to be done only once a month. The workload consisting of generating hash outputs for all coins and their Golomb encoding can be distributed among the $\sigma = 20$ proxy devices.

6. Security and Privacy Analysis

The challenge of having secure and private payment with RFID tags is due to the lack of asymmetric cryptography and blind signatures. Authentication and encryption based on symmetric keys shared by tags and readers are not suitable either, since a globally shared key would allow the adversary to jeopardize the whole system through the compromise of a single tag. Sharing a different key per tag would affect privacy. As a result, PSP's mechanism of reader authentication uses precomputed challenge-response pairs. Another challenge is due to the lack of storage space on a tag, requiring generation of coins on the fly. To detect generation of invalid coins, PSP uses Bloom filters. Finally, offline readers complicate detection of double spending. As with classical ecash protocols, PSP utilizes periodic maintenance operations and identification of users in case of double spending.

6.1. Protection against fraudulent payment

After giving a brief summary about PSP's main security concept, we formally show that, for a free rider, the best attack to invent a new coin is to exploit

the Bloom filter's false positive rate. For a malicious payee, inventing new coins is as difficult as breaking HMAC. Also, we argue why any adversary cannot or will not successfully complete reader/payee impersonation, replay of coin, and double spending attacks.

The basic idea behind PSP's two staged payment procedure is that a valid *coin* without an according *precoin* is worthless for the adversary. A simpler version of the protocol solely based on the exchange of *coin* would allow the adversary to intercept message 2 of Figure 2 and deny its delivery to the reader. The adversary would then have successfully stolen one coin he could later use for his own payment. Consequently, T_{Alice} will only send *precoin* in message 4 after it authenticated the reader in message 3. If the adversary denies delivery of message 4, he might get a valid *coin*, *precoin* pair, but he still cannot use it, as the reader has already added *coin* on its spentList after message 2. Also, the adversary cannot use this *coin* with another reader, as coins are reader dependent by using RID.

6.1.1. Inventing coins

With h being a cryptographic hash function, no adversary can "invent" coins. More precisely, it is impossible for an adversary to compute a new, valid $\{precoin, coin := h(precoin)\}$ pair that he can use to pay a reader RID. Here, "new" means that $precoin/coin$ has not yet been used at RID and is therewith based on an unused c value.

For ease of understanding and sake of clarity, operator "," as in any hash evaluation $h(a, b, c, \ldots)$ on multiple inputs $(a, b, c, \ldots)$ has been denoted as just some unambiguous pairing of inputs, cf., Section 4.5. Now, by proposing an implementation for the "," pairing and rewriting the hash evaluations, we show that PSP becomes a special case of HMAC and its security against inventing of coins provable.

An HMAC using key k on messages m of *variable* length and based on an iterated hash function h is defined as $\text{HMAC}_k(m) := h(k \oplus \text{opad}||h(k \oplus \text{ipad}||m))$, with $||$ being concatenation of inputs, and $\oplus$ is XOR. See details on message padding, ipad, and opad in [43].

Given h and a secret key k being indistinguishable from random data for an adversary, $\text{HMAC}_k(m)$ has been proven to hold the following two properties [44, 45].

1. The adversary can choose q messages $m_1, \ldots, m_q$, and query an oracle ("*oracle*") with them to get q HMACs back, i.e., $\text{HMAC}_k(m_i)$. Still, it is impossible for the adversary to find another pair $(m, \text{HMAC}_k(m))$, where $m \neq m_i, 1 \leq i \leq q$. This directly implies that the adversary cannot compute k.
2. $\text{HMAC}_k(m)$ is a pseudorandom function.

Now for PSP, we re-define the notation used in the algorithms above, thereby providing a sample implementation for the "," pairing of inputs.

(Pseudo-)Random secret key k is defined as $k := \text{ID}^{\epsilon}_{\xi^{\epsilon}}$, only known to T_{Alice} (and bank0). We define $precoin := \text{HMAC}_k(1||\text{RID}||c)$, $chall := \text{HMAC}_k(2||c)$,

$prefake := \mathrm{HMAC}_k(3||c)$, with $\{1,2,3,4\}$ publicly known constants, and $b := \lfloor \mathrm{HMAC}_k(4||c) \rfloor_1$. As c is a small value between 1 and $\gamma_{\max}$, an adversary can easily guess it. So, we also assume c to be known. Note that RID, $\{1,2,3,4\}$, and c have a fixed length. Finally, the reader's response v is defined as $v := \lfloor \mathrm{HMAC}_{K_\epsilon}(chall) \rfloor_\omega$.

The proof that PSP is as secure against inventing coins as HMAC's security now becomes straightforward:

Theorem 1. *Using the above data observed during q PSP payments, no adversary can compute a new valid precoin that is based on a valid* $(\mathrm{RID}, \mathrm{ID}, c, \epsilon)$ *tuple, where precoin has not yet been used for a payment at* RID *in epoch ϵ.*

Proof (Proof (Sketch)). If an adversary $\mathcal{A}_{\mathrm{PSP}}$ can compute a new valid *precoin* to pay at a reader RID in epoch ϵ, another adversary $\mathcal{A}_{\mathrm{H}}$, with a set of $(m_i, \mathrm{HMAC}_k(m_i))$ pairs, would be able to compute a new pair $(m, \mathrm{HMAC}_k(m))$ violating HMAC property 1. That is, if $\mathcal{A}_{\mathrm{PSP}}$ outputs a new valid $precoin'$ after observing q payments, $\mathcal{A}_{\mathrm{H}}$ outputs a new $(m', \mathrm{HMAC}_k(m'))$ pair.

We construct $\mathcal{A}_{\mathrm{H}}$ using $\mathcal{A}_{\mathrm{PSP}}$ as a subroutine in a straightforward manner, cf., Algorithm 6. (Note that for sake of simplicity, we omitted the usual probability computations of non-negligible adversarial advantages in this proof sketch.)

ALGORITHM 6: Adversary $\mathcal{A}_{\mathrm{H}}$

Choose K_ϵ;
for $i{:=}1$ **to** q **do**
  Choose $\{\mathrm{RID}_i,\, c_i, \epsilon_i\}$;
  $\mathcal{A}_{\mathrm{H}} \rightarrow \mathcal{A}_{\mathrm{PSP}} : \{\mathrm{RID}_i, c_i, \epsilon_i\}$;

  $\mathcal{A}_{\mathrm{H}} \rightarrow oracle : \{(1||\mathrm{RID}_i||c_i),\, (2||c_i),\, (3||c_i),\, (4||c_i)\}$;
  $oracle \rightarrow \mathcal{A}_{\mathrm{H}} : \{\mathrm{HMAC}_k(1||\mathrm{RID}_i||c_i),\, \mathrm{HMAC}_k(2||c_i),$
    $\mathrm{HMAC}_k(3||c_i),\, \mathrm{HMAC}_k(4||c_i)\}$;
  $\mathcal{A}_{\mathrm{H}} \rightarrow \mathcal{A}_{\mathrm{PSP}} : \{\mathrm{HMAC}_k(1||\mathrm{RID}_i||c_i),\, \mathrm{HMAC}_k(2||c_i),$
    $\mathrm{HMAC}_k(3||c_i),\, \lfloor \mathrm{HMAC}_k(4||c_i) \rfloor_1,$
    $\lfloor \mathrm{HMAC}_{K_\epsilon}(\mathrm{HMAC}_k(2||c_i)) \rfloor_\omega\}$;
end
$\mathcal{A}_{\mathrm{PSP}} \rightarrow \mathcal{A}_{\mathrm{H}} : \{\mathrm{RID}',\, c',\, \epsilon',$
  $precoin' = \mathrm{HMAC}_k(1||\mathrm{RID}'||c')\}$;
output $\{(1||\mathrm{RID}'||c'), precoin'\}$;

In Algorithm 6, $\mathcal{A}_{\mathrm{H}}$ chooses q pairs (RID, c) and asks the oracle to compute HMACs. This results in q valid PSP payment observations, i.e., *precoin*s, *chall*s, *prefake*s, *b*s, and verification bits as described before. Then these HMACs are simply forwarded to $\mathcal{A}_{\mathrm{PSP}}$, emulating q proper PSP payments. Assumed $\mathcal{A}_{\mathrm{PSP}}$ finally outputs a new valid $precoin'$ that he can use at reader RID' in epoch ϵ', then $\mathcal{A}_{\mathrm{H}}$ outputs a new valid $(m', \mathrm{HMAC}_k(m'))$ pair. $\mathcal{A}_{\mathrm{H}}$ knows that, if

precoin is valid, he only has to concatenate 1 with $(\mathrm{RID}'||c')$ to construct a new m' matching $precoin = \mathrm{HMAC}_k(m')$.

In conclusion, although any adversary might observe different RIDs during payments and might even guess c, it is impossible to compute another valid $precoin := \mathrm{HMAC}_k(1||\mathrm{RID}||c)$ as of HMAC property 1.

A rational **free rider** adversary will therefore focus on exploiting the false-positive property of readers' Bloom filters: any *invalid* coin can be accepted by a Bloom filter with probability $P = 2^{-\kappa} = 2^{-22}$. However, the free rider has to carry out such an attack *online*, by being physically close to the reader. Here, every time the free rider guesses a coin incorrectly, the reader sleeps for a reasonable amount of time (and possibly raises an alarm) such that this kind of attack quickly becomes too time consuming for the free rider. We claim that 2^{-22} probability to get *one single* coin accepted by a reader is secure enough in this scenario.

Note that in this case, getting a coin accepted at a reader does not automatically imply that this coin is really a valid coin, based on a $(\mathrm{RID}, \mathrm{ID}, c)$ tuple, but it can be a false-positive. If the reader presents such a false-positive during periodic maintenance, then bank0 can detect that this coin is in fact *not* a valid coin – using Δ.

Along these lines, a **malicious payee** adversary cannot exploit possession of a Bloom filter: while he can query his own Bloom filter offline with arbitrary *coin*s, the result of the isElement does not reveal any information whether a *coin* is a valid coin or just a false-positive. Knowing false-positives for his own Bloom filter does not help to get real coins from the bank, neither to get services from other payees. A malicious payee cannot invent coins to pay at another payee's reader.

6.1.2. Reader impersonation

One way to cheat in PSP would be to try stealing money from a (legitimate) tag. While a free rider adversary will try to impersonate as any legitimate reader, a malicious payee adversary will try to impersonate as a different payee, see Section 3.

The **free rider** initiates communication with T_{Alice}, and guesses the ω verification bits ν for *chall*. If his guess is wrong, T_{Alice} sends $prefake$ to him. If he is right, he receives *precoin* and has successfully stolen a valid $\{coin, precoin\}$ pair. However, he does not know whether he guessed correctly as he cannot distinguish whether the data he receives is for the valid coin or the fake coin. He just knows that $h(received)$ matches either *coin* or $fake$. The pair he can compute is therefore either $\{fake, prefake\}$ or $\{coin, precoin\}$. If he tries to pay with this pair, he always succeeds with probability $P_{\mathrm{steal}} = 2^{-\omega} + (1 - 2^{-\omega}) \cdot P$, triggering an alarm etc. With $\omega = 1$, the adversary can steal successfully with $P_{\mathrm{steal}} \approx 50\%$. This probability can be decreased by increasing ω at the cost of additional storage requirements on the tag. With $\omega = 1$, $\omega \cdot \gamma_{\max}$=10 byte are required. If the adversary should be able to steal a coin with only 2^{-10} probability, 100 byte of storage are required, and a probability of 2^{-32} requires

320 byte. Generally, security can be adjusted depending on the physical properties of the tag. Stealing a coin from a tag requires much more effort from the adversary than just randomly generating coins and sending them to a reader: the adversary has to be physically close to the tag to send and receive messages. Recent practical results [46] for such "skimming" attacks state distances up to 30cm between adversary and tag as realistic while using transmission powers of 4W (four times the transmission power of a GSM phone). This renders an unnoticed stealing of coins possible in only very crowded areas. Furthermore, simple countermeasures can make these attacks more difficult: as proposed by [47], [48], and implemented by tag manufacturer Alien Technologies, a simple, self-discharging capacitor on the tag can be used as a timeout mechanism. After an unsuccessful PSP protocol execution, the tag refuses another protocol execution until the capacitor is discharged. To steal more than one coin, the adversary would be required to follow the tag for an extended amount of time.

Finally, we conjecture a 50% probability of stealing *one single* valid coin to be reasonable, because the adversary does *never know* whether each single coin is a fake or not. We conjecture that triggering an alarm with 50% probability per coin and the difficulty of mounting such an attack will prevent the adversary from stealing coins in practice.

On the other hand, a **malicious payee** adversary can easily impersonate as another payee's reader, as he knows K_ϵ, i.e., the current epoch key. However, as coins are bound to RIDs in PSP, the resulting coin is useless for the malicious payee. During periodic maintenance, i.e., Algorithm 5, bank1 checks whether the reader asking for reimbursement is the same as the reader this coin was intended for.

Finally, a successful *Mafia Fraud* in PSP would be that a malicious payee successfully impersonates as another payee's reader RID to steal a valid coin from T_{Alice}. The malicious payee might now use this stolen coin to use and pay for a service provided by RID. On a side note, we again point out that Mafia Fraud is generally possible in ecash systems, but PSP can be extended, e.g., using RFID time or distance bounding protocols, see [29, 30].

6.1.3. Replay of coins

Any adversary eavesdropping payments cannot replay a coin at the same reader, because the reader stores all spent coins either in its spentList (same day), or the reader stores spent coins in its spentBF Bloom filter (after maintenance, if spentBF is still used – see Section 5.7). In any case, the reader will reject this coin. If the adversary eavesdropped a payment at a reader, he cannot replay and pay with this coin at a different reader, because coins are reader dependent. If a malicious user spends a valid coin, i.e., using an (ID, c) pair, on one reader today and re-uses this pair with another reader on *another day* after maintenance, this reader will reject the coin as it is already on its spentBF. Only if an adversary spends a valid coin on the *same day* with two *different* readers, readers cannot immediately detect cheating and will accept this coin. However, during the periodic maintenance at night, bank1 will spot this kind

of double spending, identify the malicious user using Algorithm 5, and ask for compensation.

6.2. DoS-Attacks and Reimbursement

PSP is clearly vulnerable against Denial-of-Service attacks: if any adversary repeatedly initiates communication with T_{Alice} and stops the protocol after the first message, the tag will increase counter c_i until, eventually, it cannot create new coins anymore. The tag is "exhausted" and refuses to operate until charged with money again. It is important to point out that the tag *must* increase c_i every time, because otherwise it will re-send the same *coin* on two subsequent protocol runs, therewith making it traceable. Again, we would like to point out the difficulty to mount such an attack in practice. As mentioned before, the adversary has to be physically close the tag and, using self-discharging capacitors, would need to follow the tag for an extended amount of time. This renders DoS attacks as extremely difficult. One way to furthermore mitigate DoS attacks is to increase c in Algorithm 4 *after* reader authentication, but this results in weaker privacy guarantees and will be discussed in Section 6.3.2.

Also, a free rider adversary is never able to steal money, but only to render all coins on the tag useless, i.e., a denial-of-service attack against all *coins* on T_{Alice}. As T_{Alice}'s coins are not spent, Alice can get reimbursed by the bank: the bank can verify that Alice's coins have never been added to spentBF Bloom filters. If the adversary replays $\{coin, fake\}$ pairs received from initiating communication with T_{Alice} to a reader, the reader will then add *coin* to its spentList, therewith marking this coin as "spent" in the whole system. To cope with this, each reader maintains reimburseLists. If a coin is spent, but the protocol is not successfully finished, the reader adds *coin* to its reimburseList. This allows the bank to easily reimburse Alice her money during periodic maintenance, cf., Algorithm 5. So in conclusion, Alice never loses her money.

As PSP must be non-interactive, a malicious payee might ask for more coins than Alice expects, but this has already been discussed in Section 3.2.

6.3. Privacy

As in any payment scenario, users demand their privacy. Informally, privacy subsumes *anonymity* and *unlinkability.* In PSP, anonymity denotes that no adversary, neither free riders nor malicious payees, can reveal the true identity (*name*) of a tag holder based on all the payments, information, and data the adversary observes. It must remain unclear *which* payer paid *which* payee. Unlinkability denotes that the adversary cannot link any two observed payments, so it is not possible to decide whether any two different payments originate from the same tag or not. Please refer to, e.g., [26] or [27] for formal definitions of privacy, unlinkability etc. Also note that unlinkability is stronger than *untraceability*, see [31]. Any protocol that provides unlinkability also provides untraceability, so we only focus on unlinkability.

6.3.1. Anonymity

First, note that bank0, bank1, and bank2 are three entities that must not collude as of Section 4.1. The only single entity knowing the identity of tag holders is bank0. However, the only information that bank0 receives about payments is $prechall = \{\xi, \epsilon, c\}$ provided from bank2. Bank0 does not have any access to RIDs so cannot associate any tag holder's identity to a payment made at a certain payee. Bank0 can only tell that a certain payer actually conducted some payment. All other entities, i.e., bank1 and bank2 as well as free riders and malicious payees only see payments and possibly the readers' RIDs. They cannot associate a payment at a reader to a certain tag holder, as they do not know anything about tag holders. In conclusion, PSP provides anonymity. As mentioned before, the lack of computationally expensive techniques equivalent to blind signatures implies splitting the bank into three entities as in PSP. Anonymity protection in case of a single bank entity is left as an open problem.

6.3.2. Unlinkability

Strictly according to the definition of the *strong privacy* model by [26] or [27], PSP does not guarantee unlinkability, but provides unlinkability only in the slightly weaker model by [13].

For example, according to the privacy game defined by [26], the following adversary breaks unlinkability: in the LEARNING phase, the adversary calls the SETKEY oracle to compromise $(\tau - 2)$ tags, so two tags, T_0 and T_1, remain uncompromised. He now initiates communication with T_0 a total of $\gamma_{\max}$ times, but stops protocol execution after receiving T_0's first message each time. Eventually, T_0 is "exhausted", cannot produce additional coins, and, for example, refuses to operate until recharge. Now, in the CHALLENGE phase, the adversary is presented with one of the two tags. If this tag is replying to his communication, he knows with 100% probability that it is T_1, otherwise it is T_0.

However, we claim DoS-attacks like the above exhaustion of coins in the *strong privacy* model to be unrealistic: using self-discharging capacitors together with the required physical proximity will render these attacks difficult in the real-world.

Free riders and malicious payees. In the absence of such DoS-attacks, PSP benefits from its relation to HMAC given in Section 6.1.1. Property two of HMAC directly yields that all the information sent from tags to readers, e.g., *coin*s, *chall*s, *precoin*s, looks completely random to any adversary as well as to all payees in each protocol run. More formally, as only T_{Alice} and bank0 know the HMAC's secret key $k := \text{ID}^{\epsilon}_{\xi^{\epsilon}}$, all subsequent protocol data is indistinguishable from random data for free riders and malicious payees, so linking is impossible. In the absence of DoS-attacks, PSP provides unlinkability in the model by [26].

One possibility to *cope* with DoS-based attacks would be to assume slightly weaker models of unlinkability, similar to the notions of "backward-security" by [13] and "narrow-strong-privacy" by [49]. We increase c in Algorithm 4 *after* a successful reader authentication – and not before. Therewith, above DoS-based attacks are rendered infeasible. With the [26] model, tags now would become linkable, as they will send the same *coin* and $fake$ on two subsequent, but

unfinished payments. The adversary can therefore link two payments if he sees twice the same $coin$, $fake$ pair. This attack requires that a tag does not have any successful protocol execution in between the two interactions of the adversary. As this might be unrealistic in many real world situations, along the lines of [13] and [49], *one* successful protocol interaction between two single adversarial interactions is allowed. As PSP's payment data looks (pseudo-)random to free riders and malicious payees, PSP then provides unlinkability according to the models of [13, 49].

It is worthwile noting that unlinkability is achieved solely within tags of the *same epoch*, and not among all tags in the system. The reason for that are the ϵ values that are disclosed at each tag communication. If an adversary sees two successful payments with different ϵ, e.g., once with ϵ_i and once with ϵ_{i-1}, he can decide that these two payments were from different tags. However, we argue that the set of tags within an epoch is large enough, as it is about half the total number ($2\eta = 10^7$) of tags, such that this does not cause significant linkability issues.

The bank. First, note that bank0, bank1, and bank2 do not have any temporal link between coins received during maintenance and the payments conducted, because readers only connect during certain periods. While bank0 and bank2, using ξ as part of $prechall$, are able to determine that some payer did some payments, there is no way for them to link any real coins or payments. Finally, bank1 sees coins, but it cannot distinguish them from random data. In conclusion, PSP provides unlinkability.

7. Conclusion

Secure, privacy-preserving, offline electronic payments only using tiny RFID tags is a new and challenging problem. In this paper, we presented PSP, a solution minimizing computational requirements for the tag, but still offering protection against overspending and privacy against payees and the bank. Adversaries cannot invent new coins, replay, or steal coins from legitimate users of the system. Multiple different, untrusted payees, e.g., a metro, a toll road system or just vending machines, cannot trace or link subsequent transactions of users to the same tag. Privacy is assured. With PSP, tags are only supposed to evaluate a hash function and store 84 byte in non-volatile memory. Finally, readers can be offline most of the time and connect only rarely for synchronization.

References

[1] Transport for London, Oyster online, `https://oyster.tfl.gov.uk/oyster/entry.do` (2009).

[2] D. Chaum, Blind signatures for untraceable payments, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 1982, pp. 199–203.

[3] D. Chaum, A. Fiat, M. Naor, Untraceable electronic cash, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 1988, pp. 319–327, ISBN 3-540-97196-3.

[4] S. Brands, Untraceable off-line cash in wallets with observers, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 1993, pp. 302–318, ISBN 3-540-57766-1.

[5] S. Micali, R. Rivest, Micropayments revisited, in: RSA conference, San Jose, USA, 2003, pp. 149–163, ISBN 3540432248.

[6] R. Rivest, Peppercoin micropayments, in: Financial Cryptography, Key West, USA, 2004, pp. 2–8, ISBN 3-540-22420-3.

[7] H. van Tilborg (Ed.), Encyclopedia of Cryptography and Security, Springer Verlag, 2005, ISBN 038723473X.

[8] G. Avoine, E. Dysli, P. Oechslin, Reducing time complexity in rfid systems, in: Selected Areas in Cryptography, Kingston, Canada, 2005, pp. 291–306, ISBN 9783540331087.

[9] S. Weis, S. Sarma, R. Rivest, D. Engels, Security and privacy aspects of low-cost radio frequency identification systems, in: Security in Pervasive Computing, Boppard, Germany, 2003, pp. 201–212, ISBN 3-540-20887-9.

[10] G. Tsudik, Ya-trap: yet another trivial rfid authentication protocol, in: Int. Conference on Pervasive Computing and Communications Workshops, Pisa, Italy, 2006.

[11] Y. Choi, M. Kim, T. Kim, H. Kim, Low power implementation of sha-1 algorithm for rfid system, in: Tenth Int. Symposium on Consumer Electronics, St. Petersburg, Russia, 2006, pp. 1–5, ISBN 1-4244-0216-6.

[12] R. Di Pietro, R. Molva, Information confinement, privacy, and security in rfid systems, in: Lecture Notes in Computer Science, Volume 4734, 2007, pp. 187–202, ISBN 9783540748342.

[13] T. Dimitrou, rfiddot: Rfid delegation and ownership transfer made simple, in: Int. Conference on Security and privacy in Communication Networks, Istanbul, Turkey, 2008.

[14] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, P. Sobalvarro, The millicent protocol for inexpensive electronic commerce, in: World Wide Web Conference, Boston, USA, 1995.

[15] R. Rivest, A. Shamir, Payword and micromint–two simple micropayment schemes, in: Int. Workshop on Security Protocols, Paris, France, 1997, pp. 69–87, ISBN 3-540-64040-1.

[16] F. Garcia, G. Gans, R. Muijrers, P. Rossum, R. Verdult, R. W. Schreur, B. Jacobs, Dismantling mifare classic, in: European Symposium on Research in Computer Security, Malaga, Spain, 2008, pp. 97–114, ISBN 978-3-540-88312-8.

[17] Alien Technology, Rfid tags, `http://www.alientechnology.com/tags/index.php` (2009).

[18] Visa USA, Visa paywave, `http://usa.visa.com/personal/cards/paywave/index.html` (2009).

[19] MasterCard, Paypass, `http://www.paypass.com/` (2009).

[20] T. Heydt-Benjamin, D. Bailey, K. Fu, A. Juels, T. O'Hare, Vulnerabilities in first-generation rfid-enabled credit cards, in: Finanical Cryptography and Data Security, Scarborough, Trinidad, 2007, pp. 2–14, ISBN 9783540773658.

[21] Wired, Mcdonald's tries out new rfid-enabled pay-by-phone coupons, `http://blog.wired.com/gadgets/2008/05/mcdonalds-tries.html` (2008).

[22] G. Venkataramani, S. Gopalan, Mobile phone based rfid architecture for secure electronic payments using rfid credit cards, in: Int. Conference on Availability, Reliability and Security, Vienna, Austria, 2007.

[23] InformationWeek, Visa debuts rfid-enabled card-payment system, `http://www.informationweek.com/news/mobility/RFID/showArticle.jhtml?articleID=60403344` (2005).

[24] InformationWeek, Rfid helps feed parking meters, `http://www.informationweek.com/news/mobility/RFID/showArticle.jhtml?articleID=174900727` (2005).

[25] A. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, Internet Mathematics 1 (4) (2003) 485–509, ISSN 1542-7951.

[26] A. Juels, S. Weis, Defining strong privacy for rfid, in: PerCom Workshops, White Plains, USA, 2007.

[27] S. Vaudenay, On privacy models for rfid, in: ASIACRYPT, Kuching, Malaysia, 2007, pp. 68–87, ISBN 978-3-540-76899-9.

[28] Y. Desmedt, C. Goutier, S. Bengio, Special uses and abuses of the fiat-shamir passport protocol, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 1987.

[29] G. Hancke, M. Kuhn, An rfid distance bounding protocol, in: Int. Conference on Security and Privacy for Emerging Areas in Communications Networks, Athens, Greece, 2005.

[30] G. Hancke, Practical attacks on proximity identification systems, in: Symposium on Security and Privacy, Oakland, USA, 2006, pp. 328–333, ISBN 0-7695-2574-1.

[31] C. Chatmon, T. van Le, M. Burmester, Secure anonymous RFID authentication protcols, Tech. rep., Florida State University, Department of Computer Science, Tallahassee, Florida, USA, http://www.cs.fsu.edu/~burmeste/TR-060112.pdf (2006).

[32] The Guardian, Oyster data use rises in crime clampdown, http://www.guardian.co.uk/technology/2006/mar/13/news.freedomofinformation (2006).

[33] E.-O. Blass, A. Kurmus, R. Molva, T. Strufe, Psp: Private and secure payment with rfid, in: 8th ACM workshop on Privacy in the Electronic Society, Chicago, USA, 2009.

[34] C. Lim, T. Kwon, Strong and robust rfid authentication enabling perfect ownership transfer, in: Conference on Information and Communications Security, Raleigh, USA, 2006.

[35] A. Shamir, Squash — a new mac with provable security properties for highly constrained devices such as rfid tags, in: Fast Software Encryption (FSE), Lausanne, Switzerland, 2008.

[36] Greater London Authority, Mayor to give away 100,000 free oyster cards, http://www.london.gov.uk/view_press_release.jsp?releaseid=11611 (2007).

[37] Transport for London, London travel report 2007, http://www.tfl.gov.uk/assets/downloads/corporate/London-Travel-Report-2007-final.pdf (2008).

[38] Transport for London, Key facts, http://www.tfl.gov.uk/corporate/modesoftransport/londonunderground/1608.aspx (2009).

[39] Transport for London, London busses, http://www.tfl.gov.uk/corporate/modesoftransport/1548.aspx (2009).

[40] NXP Semiconductors, Mifare 1k, http://mifare.net/products/smartcardics/mifare_standard1k.asp (2009).

[41] F. Putze, P. Sanders, J. Singler, Cache-, hash- and space-efficient bloom filters, in: Workshop on Experimental Algorithms, Rome, Italy, 2007, pp. 23–36, ISBN 978-3-540-72844-3.

[42] Amazon, Webservices simple monthly calculator, http://calculator.s3.amazonaws.com/calc5.html (2009).

[43] H. Krawczyk, M. Bellare, R. Canetti, Hmac: Keyed-hashing for message authentication, RFC 2104, http://www.ietf.org/rfc/rfc2104.txt (1997).

[44] M. Bellare, R. Canetti, H. Krawczyk, Keying hash functions for message authentication, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 1996.

[45] M. Bellare, New proofs for nmac and hmac: Security without collision-resistance, in: Annual Int. Cryptology Conference, Santa Barbara, USA, 2006.

[46] G. Hancke, Practical eavesdropping and skimming attacks on high-frequency rfid tokens, Journal of Computer Security (to appear), `http://www.rfidblog.org.uk/Hancke-JoCSSpecialRFIDJune2010.pdf` (2010).

[47] A. Juels, Yoking-proofs for rfid tags, in: PerCom Workshops, Orlando, USA, 2004, pp. 138–143, ISBN 0-7695-2106-1.

[48] M. Burmester, B. de Medeiros, R. Motta, Provably secure grouping-proofs for rfid tags, in: Int. Conference on Smart Card Research and Advanced Applications, London, UK, 2008, pp. 176–190, ISBN 978-3-540-85892-8.

[49] A. Sadeghi, I. Visconti, C. Wachsmann, Anonymizer-enabled security and privacy for rfid, in: Conference on Cryptology And Network Security, Kanazawa, Japan, 2009.