International Journal on Very Large Data Bases manuscript No.
(will be inserted by the editor)

Succinct Representation of Flexible and Privacy-Preserving Access Rights[⋆]

Marina Blanton, Mikhail Atallah

Department of Computer Science
Purdue University
West Lafayette, IN 47907
e-mail: {mbykova,mja}@cs.purdue.edu

The date of receipt and acceptance will be inserted by the editor

Abstract We explore the problem of portable and flexible privacy preserving access rights that permit access to a large collection of digital goods. *Privacy-preserving* access control means that the service provider can neither learn what access rights a customer has nor link a request to access an item to a particular customer, thus maintaining privacy of both customer activity and customer access rights. *Flexible* access rights allow a customer to choose a subset of items or groups of items from the repository, obtain access to and be charged only for the items selected. And *portability* of access rights means that the rights themselves can be stored on small devices of limited storage space and computational capabilities such as smartcards or sensors, and therefore the rights must be enforced using the limited resources available.

In this paper, we present and compare two schemes that address the problem of such access rights. We show that much can be achieved if one allows for even a negligible amount of false positives — items that were not requested by the customer, but inadvertently were included in the customer access right representation due to constrained space resources. But minimizing false positives is one of many other desiderata that include protection against sharing of false positives information by unscrupulous users, providing the users with transaction untraceability and unlinkability, and forward compatibility of the scheme. Our first scheme does not place any

[⋆] Portions of this work were supported by Grants IIS-0325345, IIS-0219560, IIS-0312357, and IIS-0242421 from the National Science Foundation, Contract N00014-02-1-0364 from the Office of Naval Research, by sponsors of the Center for Education and Research in Information Assurance and Security, and by Purdue Discovery Park's e-enterprise Center.
Parts of this work appeared in the proceedings of ESORICS 2004 and SACMAT 2005.

constraints on the amount of space available on the limited-capacity storage device, and searches for the best representation that meets the requirements. The second scheme, on the other hand, has (modest) requirements on the storage space available, but guarantees a low rate of false positives: With $O(mc)$ storage space available on the smartcard (where m is the number of items or groups of items included in the subscription and c is a selectable parameter), it achieves a rate of false positives of m^{-c}.

Key words Compact representation – privacy-preserving access rights – flexible access rights

1 Introduction

The focus of this work is on the specification of access rights that permit privacy-preserving access to a large collection of digital goods (e.g., articles, books, magazines, multimedia objects, or any other type of digital data items). With a large number of subscription-based services available today, customers would be more willing to use such services if we could guarantee that access to the digital goods is anonymous and their preferences and access patterns cannot be tracked. That is, if customers can purchase their subscription anonymously (either by authenticating using an anonymous authentication scheme, or by purchasing the card anonymously from a public bookstore or cyber-café) and further interaction with the server does not reveal customer or card-specific information while still allowing access to the authorized set of digital goods, then customer privacy is guaranteed.

1.1 Motivation

During the confirmation hearings of a nominee for the U.S. Supreme Court back in 1988, the issue of which movies he had rented came forth. The records of which movies he had actually rented, had been obtained by a local Washington newspaper from a video rental store, by simply asking the store for them. Today the Video Privacy Protection Act (18 U.S.C. 2710) prevents video stores from releasing such information without the customer's written consent, but it is nevertheless all too easy for such records to be released nevertheless, either inadvertently or through a break-in, spy-ware, insider misbehavior (rogue employees), social engineering, etc. Moreover, the video-privacy bill is specifically about movie rentals (it does not cover, e.g., magazine and other subscriptions). The problem is exacerbated in the online word, as it is then possible for a server to determine not only which material the customer accessed, but also how many times, when, for how long each time, etc. For example, a customer who accesses much material about a disease runs the risk of an inference being made about her having that disease (or having a lifestyle that puts her at risk for it). Some recent encryption-based digital-rights management technologies not only reveal which encrypted material was downloaded, but also the exact times at which the user chose to view the downloaded material (as each viewing requires that the server sends a key to the client-end viewing software that is entrusted by the content-owners with decrypting/displaying the material and then destroying the key). These technologies are used for such purposes as protecting the revenue-streams of content-owners against piracy, allowing corporations to enforce policies on documents and emails without fear of employee non-compliance (e.g., to remotely shred an old document or email, the server simply deletes the key associated with it, and the employees' hard drives are left with unusable encrypted material). Many of the technologies that have been deployed, or are under development, have chilling privacy implications. Even as they have such large potential for damage to privacy, these techniques have largely failed to prevent piracy, as they are typically defeatable by a determined attacker. This is why hardware-based digital rights management techniques are being deployed (they are much harder for an attacker to crack than purely software-based ones) even though they could enable more stealthy ways for software publishers to spy on users, to know what is on a user's computer, to control what the user can and cannot execute, view, connect to the computer, print, etc. Although our schemes use tamper-resistant cards, they do not harm user privacy because at no time does a card "know" who the customer is.

The customer and regulator complaints (including lawsuits) filed against a major clickstream-information collecting company, alleging what amounts to cyber-spying, is but one example illustrating people's sensitivity when it comes to the tracking of their online activities (even for apparently innocuous marketing purposes). These fears may be well justified, because history is full of examples where information collected for a benevolent purpose was subsequently used for nefarious purposes (even prior to the cyber-age, e.g., the Dutch government records that listed their citizens' religion were subsequently used by the Nazis for a horrendous purpose). The misuse need not come from the data-collecting entity: The data may simply fall in the wrong hands through a security breach – the last few years have seen an avalanche of security breaches in which private information was seriously compromised.

In view of the above, a customer's trust that the data collectors will not misuse the data is only a first line of defense (and a rather flimsy one, based on the evidence). Privacy-conscious customers who find appealing a "defense in depth" that protects them from such possible mis-haps, can take steps to avoid revealing their identity to the server by, e.g., obtaining access through anonymizing proxies, or simply from a cyber-café. This, however, does not work well in the context of a for-pay access to online material, that typically requires the server to learn the identity of the subscriber through (e.g.) the entry of a login and password tied to the subscriber's real identity through the subscription information (typically including the name, email address, and credit-card payment information). The need exists for schemes, such as presented in this paper, that support for-pay subscriptions without compromising subscriber privacy, yet while preserving the content-owner's rights.

1.2 The framework

As the number and the level of maturity of services that offer access to digital goods grow, the level of flexibility of such systems will also grow. To make access as convenient to the customers as possible, such systems might allow customers to subscribe to items of their choice with fine granularity, and not limit their choice to a small number of predefined subscription types. In the most flexible setting, the system allows each customer to select a set of objects to which they wish to have access and correspondingly pay for. Depending on the structure of the data repository, a customer may be able to select individual items or groups of items based on their type, topic, or another classification scheme. The customer then receives an access policy configuration that is unique to her subscription request.

With this model in place, the service provider can no longer store a complete description of customer access rights at his end, because if he did, it would violate privacy requirements. A solution is to store access rights at the customer end using tamper-resistant devices such as smartcards. The main challenge is then to design a scheme that would permit the customer to access the goods to which they have subscribed and at the same time preserve their anonymity by making their transactions untraceable and unlinkable[1].

Since customer policy configuration is stored on weak devices such as smartcards, such devices are normally limited in their computational power and storage space, especially if their cost must be kept low (which is the case, for instance, with short-term orders and/or disposable cards). Limited resources, however, conflict with our intent to provide flexible access to the items of customer choice, if the size of the data collection is very large. There is therefore a need for techniques that permit succinct representation of customer rights and avoid the use of expensive computations.

If a card that stores a customer's subscription set does not have enough capacity to store at least one bit per item in the (potentially huge) data repository, then it becomes impossible for it to exactly represent all possible subsets of the repository items. Thus, some items or subset of items will have to share the same configuration and introduce "false positives" into the scheme — a *false positive* is an item that was not listed in the subscription, but which the customer is permitted to access. This model is acceptable if the probability of a false positive (PFP) is small enough. A major goal is then to design a scheme for computationally efficient access control enforcement under space constraints that minimizes the number of false positives

[1] It is a consideration that privacy may be lost, if a dishonest customer attempts to misuse the system by, for instance, breaking the card and distributing its contents to a large number of people, but for honest subscribers access will always be anonymous.

implicit to each card. Of course, false negatives are not tolerated: a customer who has paid to subscribe to an item must always be granted access to that item. Minimizing false positives is not the only requirement: others include protection against sharing of false positives information by unscrupulous users, providing the users with transaction untraceability and unlinkability, and forward compatibility of the scheme; these, and other design goals, are stated in more detail later in the paper.

1.3 Counter-indications

Our schemes are suitable for the realm of digital-rights management, in situations where a false-positive access to a document or a music is tolerable if it has a reasonably low probability of occurrence *and* the negative consequences would be tolerable if it were to occur. Both conditions are important because, even when the probability (call it p) of a false positive is reasonably low, if the damage (call it d) from a false positive is large enough, the expected damage $p * d$ may be unacceptably high in some situations. In view of the above, the schemes described in this paper should not be used in situations where a false positive has a catastrophic consequence, such as unauthorized access to a patient's medical records, or the software that controls a power plant's machinery.

But even when the probability of false positives is close to zero, and/or when the apparent and measurable damage from such a false positive is zero, there are situations where privacy-preserving techniques are not recommended (*any* of them, not just the ones we propose). These are situations where even legitimate (not just false positive) accesses could be a cause for concern if they follow certain patterns or are done by certain individuals. Specifically, privacy-preserving techniques are not appropriate if the legitimately accessed material is of such a nature as to inherently require monitoring or auditing by law-enforcement agencies. For example, if the on-line material is restricted-access because of its possible use to evil-doers intent on acts of violence, financial fraud, or disruption, then an audit trail of which authorized individuals accessed it (and when they accessed it) is needed by law-enforcement agencies to determined (e.g.) who "leaked it out" in an unauthorized fashion. The use of privacy-preserving techniques becomes obviously problematic in such a framework, as it would serve to protect the culprit. The framework we have in mind for our schemes is therefore more one involving commercially valuable but innocuous content, such as music, movies, e-books, databases of past sports events and data, historical trading transactions (stocks, bonds, commodities), and other specialized databases that are unlikely to be of use to criminal elements.

1.4 Our contributions

Our contributions are as follows: we give two solutions that address the problem of flexible and privacy-preserving access rights. Our first solution utilizes random permutations and does not place any constraints on the amount of storage space that should be available on the smartcard. Instead, it searches for the best solution that meets the space requirements and satisfies the service provider. The second solution is based on the use of minimal perfect hash functions (MPHF), and differs from the previous solution in that it is guaranteed to result in a low rate of false positives for any subscription order, but uses storage space proportional to the subscription size. More precisely, given a smartcard with $O(mc)$ storage space, where m is the number of items or groups of items included in the subscription and c is an arbitrary parameter, this scheme achieves the rate of false positives m^{-c}. For both schemes, we provide solutions for (i) “flat” data repositories, where the collection of documents is not organized into a data structure; and (ii) hierarchically structured collections of data items such as trees.

1.5 Organization of the paper

The rest of this document is organized as follows: In section 2, we review prior related literature. Section 3 gives a more precise problem description and lists design goals. In section 4, we describe our first, permutation-based approach for both unstructured and structured data repositories and provide its analysis. In section 5, we give the second, minimal perfect hash function based approach, its analysis, and extensions. Section 6 compares the schemes and concludes the paper.

2 Related Work

Work conducted on XML explores the problem of access control for online data repositories, which includes securing access to XML documents and using XML as a tool for specifying security policies (see, e.g., [6–8, 18, 19]). Bertino et al. [5] use binary strings to represent both customer policy configurations and document policies, but they allocate one bit per policy on the assumption that there will be a limited number of different subscription types. Thus, their approach becomes inefficient as the data repository grows in size and each customer chooses a customized document subscription set. The topic of digital libraries is also related to this work, but literature on digital libraries usually does not address access control.

The idea of achieving space efficiency at the cost of a small probability of false positives was introduced in Bloom [9]. Bloom filters support approximate membership queries and are widely used in a broad spectrum of applications ([12, 20, 27], to name a few). Such data structures achieves a better space utilization than simple hash representation, but the filter length (which in our case corresponds to the card capacity) still should be larger than the total number of items in the set to result in a reasonable performance. This is not suitable for cards of small capacity, and even customized Bloom filters do not appear to provide acceptable results.

Other techniques for concise representation of portable access rights were used in the context of software license management [4,1]. These solutions, however, do not apply to our problem, mainly because we cannot afford to avail ourselves of resources external to the card (as was the case in [4,1]). The more recent work in [13], on the other hand, considers the same problem of portable and flexible access rights for large data repositories. In [13], the authors consider static policy assignment to all repository documents, which makes addition of new items problematic without performing periodic policy updates (after which all smartcards must be refreshed) and also makes it possible for dishonest users to share and use information about false positives.

Some of our solutions use minimal perfect hash functions (MPHF) as their underlying building blocks. MPHFs have received significant attention, and a number of algorithms can be found in [23,22,17,21]. There are MPHFs and order-preserving MPHFs (OPMPHFs) that for random m strings take the total of $O(m)$ bits to store the functions (and this is also the lower bound). See [23,21] for more detail.

Work on unlinkability and untraceability was started by Chaum [16] and received significant attention in recent years. In particular, work on unlinkability includes anonymous group authentication ([2,11,15,24–26,28–30] and others) and unlinkable serial transactions [31] for subscription-based services. Prior work, however, does not account for the fact that descriptions of access rights (or service types) may be long and required to be portable, while we describe schemes that combine compact policy representation with transaction unlinkability.

3 Problem Specification

3.1 General model

The general model used in our work is depicted in Figure 1 and consists of two stages. During the *initialization stage* — which can take place in a bookstore, at a public library, or at home — a customer chooses items of his choice, pays for the items selected, and receives a customized card that subsequently permits access to these items. During the *card usage stage* — which can likewise be done from a home computer, library, etc. — the customer can request access to any items from the repository. If the card permits access, it uses the built-in anonymous authentication protocol to prove its authenticity to the server and then obtains the item from the server.

Here, by "server" we do not necessarily mean a remote server. Instead, it could be a local (trusted) content player at the client end or any other mechanism used by the content owners to enforce their policies. In that case, the encrypted content is already stored at the client's end and the server grants access by decrypting and then displaying it. Therefore the model does not necessarily assume network connectivity for data access.

Initialization

1. Customer chooses m items for purchasing.
2. Server processes the order and creates a card.
3. Customer pays and receives the card.

Operation

	Card		**Server**
1.	Receives from the user a request to access item i		
2.	Checks if access can be granted		
3.	Anonymously authenticates to the server and requests item i on behalf of the user	$\longrightarrow$	Verifies authentication credentials
4.		$\longleftarrow$	Sends item i to the requester

Fig. 1 General model of operation.

Throughout this paper, we assume that a card is authentic and can *anonymously* and at low computational cost authenticate itself to the server. A number of solutions that range from trivial secret key schemes to more complex and provably secure schemes (e.g., [2,25]) can be used to achieve this goal. Card unforgeability is achieved through other, standard techniques described in prior literature and is out of scope of this work.

As an example of a provably secure scheme that allows users to anonymously access the service, we show how group signatures (e.g., [2,3,10,14]) can be used to achieve this goal. A *group signature* is a cryptographic construction that allows a member of the group to sign messages anonymously on behalf of the group. In case of a dispute, however, the identity of the originator of such a signature can be revealed by a designated authority (called group manager). The interactive version of group signatures (what is needed here) is called *identity escrow*, and each group signature scheme can be used in the interactive mode. In such schemes, there is a protocol that allows a user to join the group and become its member (which in our case will be done at the subscription time). As a result, user credentials for the group signature scheme and her access rights that permit access to the documents in the repository will be written on a card. Then every time a user wants to access an item at the server, she will first anonymously authenticate to the server using her group signature credentials and then invoke her access rights to obtain access to the document of her interest.

3.2 Notation

In the rest of this paper we use the following notation: the data repository contains n items numbered 1 through n. A customer can request access to (and accordingly pay for) m items, $1 \leq m \leq n$. Access rights are stored on a card of limited capacity of k bits, where $k < n$ and $k < m \log n$.[2]

We use the term *order* to refer to a subscription order of m documents for which the customer pays and receives a card that permits access to those documents. We use the term *request* to refer to a request to access a document by a customer who already possesses a card and wishes to view a document. A customer subscription order of m items is denoted as $\{i_1, \ldots, i_m\}$, where i_j uniquely identifies a single document in the repository and $1 \leq i_1 < \ldots < i_m \leq n$.

3.3 Design goals

The design goals that we require any solution to have are as follows:

Low rate of false positives. The probability (or rate) of a false positive (PFP) — the probability that a random document not in the set of m subscription documents is among the documents to which access is authorized — is the main evaluation criterion of any approach, and the goal of this work is to minimize such a PFP. The PFP obviously depends on the storage space available on the card.

Transaction untraceability and unlinkability. For customer privacy, we require that after a customer buys an access card and uses it to retrieve an item from the repository, it is not possible to use the data sent in the request to tell with probability significantly greater than a random guess which customer is making this request. Similarly for transaction unlinkability, we require that given two access requests it is not possible to tell with probability significantly greater than a random guess whether these two requests originated with the same user.

Unique policy representation ("no sharing of false positives"). It is also a design requirement that every policy representation stored on a card is unique. More precisely, given two subscription requests that contain identical sets S_1 and S_2 of items to be purchased, their representations stored on access cards C_1 and C_2 will be different and the false positives implicit to each card will also be different. We require this property in order to eliminate the possibility of sharing information about false positives by dishonest customers. When this is not the case and a fixed set of items triggers the same set of false positives, dishonest users might share this information through public channels such as the Internet, making the scheme unusable for the data provider.

Note that this will prevent sharing of information about false positives, but not all possible forms of information sharing. That is, content sharing by dishonest users is always possible regardless of an access control mechanism used. Our goal here is to prevent sharing of information

[2] If $k \geq m \log n$ then the card can explicitly store the m items. So we henceforth assume that k is less than $m \log n$, i.e., that space on the card is tight.

about access rights and, more importantly, information about illegitimate access rights that the user should not possess. Access rights information is much more convenient to share than voluminous contents.

No additional sources of information. The schemes we design are for online data repositories that, using a card, can be accessed from a number of places such as terminals at public libraries, bookstores, home workstations, and other places. Therefore, if a scheme were to require some additional information to be stored on external storage, in our scenario there is no reasonable place at which such information could be stored (and, as was mentioned above, no user information can be stored at the server itself by the untraceability requirement). Thus, the access card itself should contain all information necessary to perform access verification.

Fast access verification, fast card generation time. These parameters also serve as evaluation criteria of each scheme, and in general we require card generation time to be bounded by a low-degree polynomial in n or, preferably, by a polynomial in m. Access verification time should be bounded by $O(k)$, where k is the space available on the card, because each card is assumed to be a computationally weak device.

Forward compatibility. In any proposed solution, if a card is created at time t_1 when the data repository contained n_1 documents, it also should stay operational at time $t_2 > t_1$ when the data repository contains $n_2 > n_1$ documents. In other words, the scheme should remain operational as new documents are added to the data repository.

Another important feature of a scheme that produces access right representations is **Support for dynamic changes to the repository**. Namely, every time changes to the data repository happen, previously issued cards with user access rights remain functional on the modified version of the data repository. Note that the forward compatibility requirement above partially covers this feature, but, for instance, deletions from the repository are not addressed. Other changes to the repository include restructuring of the items currently present in the repository. This, however, is applicable only to structured data collections such as, e.g., hierarchies and can be handled solely using support for additions and deletions. In our design requirements we do not strictly require support for document deletions from the repository, because the scheme can be used regardless of availability of this feature. That is, if deletions are not handled automatically, the repository can be periodically "cleaned" (e.g., once a year), removing unwanted documents all at once. We, however, analyze our solution with respect to such support and show how dynamic changes can be handled in each of the schemes. It is also worth mentioning here that a document cannot be removed from the data repository while there is at least one customer with valid access rights to that document (i.e., documents are not likely to be removed often).

Note that the above requirements make our problem very different from mere data compression. Another difference from data compression is that here each representation on the card must be usable "as is" without uncompressing it first: There is no room in the card for decompression, and using server memory for decompression would reveal enough about the card to make profiling of the card's usage patterns possible (recall that contents of two cards are different even if both of them contain the same subscription set. They are in some sense an implicit ID for the card and should therefore not be revealed to the server). Client memory is not suitable for decompression either because it cannot be trusted.

4 Permutation-based Approach

In this section we present our first solution. We first describe our approach to an unstructured collection of documents and provide its analysis. Structured collections of documents are addressed in section 4.7.

Recall that the card's capacity is k bits. In the rest of this section we assume that those k bits are divided into ℓ slots of $\log n$ bits each and therefore the card's capacity is $k = O(\ell \log n)$.

Our solution consists of generating random permutations of the documents included in an order until they are clustered in such a way that the cost (in terms of false positives) of storing the permuted documents on a smart card is below a certain threshold (defined later). After generating a permutation of the documents, we run an evaluation algorithm to compute the cost of the optimal solution for that particular set of permuted documents. If the cost is acceptable, the algorithm terminates and the solution is written to the card; otherwise, a new permutation is generated and tested. The information written on the card includes data that can be used to reproduce the permutation, as well as a number of document intervals that indicate access to which documents should be granted. The intervals include all documents from the subscription order and as few additional documents as possible.

Consider an oversimplified example where the repository has the size of 20, our card can store 2 intervals, and we receive a customer subscription order for documents 1, 5, 7, 9, 13, and 19. Suppose that after permuting the documents we obtain set $\{2, 3, 4, 15, 16, 18\}$, so the best option in this case is to use intervals 2–4 and 15–18 for storing the set on the card. The cost of a solution is computed as the number of false positives, and in this case the cost of the permutation is equal to 1.

Both the random permutation seed and the document intervals are subject to the card's storage constraints. Since the smartcard's capacity is $O(\ell \log n)$, we can use it to store $O(\ell)$ numbers within the range $\{1, \ldots, n\}$, or ℓ intervals. The permutation seed can also be up to $O(\ell \log n)$ bits long.

Every interval included in a solution can be either *positive*, i.e., specifies a range of documents to which access should be granted, or *negative*, i.e., specifies a range of documents to which access should be denied. In the case of unstructured data (i.e., where the data repository is a mere collection of numbered items, not organized into a hierarchy or any other type of data structure), negative ranges do not improve the result by decreasing the cost of a solution, as the lemma below shows (we, however, show later that they are necessary for structured data).

Lemma 1 *For unstructured data, for every solution of cost C expressed using both positive and negative ranges there is a solution of cost C' expressed using only positive ranges, such that $C' \leq C$.*

Proof See Appendix A.

We first present an algorithm for producing a suitable encoding to be placed on a card (section 4.1). This is a high level algorithm that tries different solutions until the conditions corresponding to the policies are satisfied. It uses two additional algorithms as its subroutines: an algorithm to produce a permutation (section 4.3) and a linear-time algorithm to compute a cost of a permutation (given in section 4.2). We give asymptotic bounds of our solution and also discuss possibilities for generating a random permutation. Later in this section we explore this approach in terms of its economic feasibility (section 4.5), and also provide an extension that covers structured data (section 4.7).

4.1 Algorithm for producing a solution

To find a suitable encoding for a customer order, we might have to try numerous permutations of n elements until one that satisfied certain criteria is found. These criteria can be expressed in terms of the cost of a solution (e.g., the number of false positives for the permutation produced falls below a certain threshold), in terms of a time interval during which a solution should be computed, or some other requirements. These rules are examined in more detail in section 4.5.

The algorithm we provide below takes a subscription order of m documents and a set of rules τ that tell the algorithm to stop when they are satisfied. It runs until a suitable solution is found and returns an encoding to be stored on a smartcard, which consists of a permutation seed s and ℓ intervals that optimally represent the documents $\{i_1, \ldots, i_m\}$.

Input: The repository size n, a customer order of m documents $\{i_1, \ldots, i_m\}$, and a set of stopping criteria $\tau = \{\tau_1, \ \ldots, \ \tau_t\}$.

Output: A seed s for generating a permutation and ℓ intervals to be stored on the smartcard.

Algorithm 1:

1. Seed the permutation algorithm with a random number s.
2. Permute the m documents to get $p_j = \pi_s(i_j)$ for each document $i_j \in \{i_1, \ldots, i_m\}$.
3. Sort the p_i's ($O(m \log(m))$ time).

4. Run the evaluation algorithm to find the cost of the permutation ($O(m)$ time, per section 4.2).
5. Apply the evaluation rules τ to the result: if a sufficient subset $\tau' \subseteq \tau$ of them, $1 \leq |\tau'| \leq t$, is satisfied, output the solution. Otherwise, go to step (1).

The asymptotic bound of a single run of this algorithm depends on the choice of the permutation function (discussed in section 4.3). The total running time of the algorithm depends on the evaluation criteria and cannot be expressed as a function of the input parameters in the general case. The upper bound of the algorithm is $O(n^{\ell})$ loop invocations, but typical values are lower. This time is constrained by the space available for storing a random seed s: there are $O(2^{\ell \cdot \log n}) = O(n^{\ell})$ possible seed values that can be stored on the card.

4.2 Algorithm for computing the cost of a permutation

The algorithm given in this section corresponds to step 4 of Algorithm 1. As the input, it expects a set of m distinct permuted documents sorted in increasing order $p = \{p_1, \ldots, p_m\}$ and computes ℓ disjoint intervals of the minimal cost that include all of the p_i's and as few other documents as possible. Our algorithm works by computing distances between the documents in the set p and excluding the largest $\ell - 1$ of them, so that the overall cost of the covering is minimized.

Input: The repository size n and a sorted set of m elements $p = \{p_1, \ldots, p_m\}$.

Output: ℓ disjoint intervals that contain all of the p_i's and as few other elements as possible.

Algorithm 2:

1. Let x be the value of p_1, y the value of p_m. Compute $c_1, \ldots, c_{m-1}$, where c_i is the number of documents between the elements p_i and p_{i+1} not including either p_i or p_{i+1}. That is, $c_i = p_{i+1} - p_i - 1$.
2. In $O(m)$ time select a $(\ell - 1)^{\text{th}}$ largest among $c_1, \ldots, c_{m-1}$ (say it is c_j).
3. In $O(m)$ time go through $c_1, \ldots, c_{m-1}$ and choose $\ell - 2$ entries that are $\geq c_j$. Those entries and c_j correspond to the $\ell - 1$ "gaps" between the optimal k intervals, i.e., they define the optimal ℓ intervals.

Note that the "cost" of the solution is $C = c_1 + ... + c_{m-1} -$ (sum of the largest $\ell - 1$ c_i's), which also proves the correctness of the algorithm because $c_1 + \ldots + c_{m-1}$ is the number of documents between positions x and y other than the elements of p, and the best that can be done is by "excluding" the large c_i's from the chosen intervals. It is also clear that the algorithm runs in $O(m)$ time, since every step (1)–(3) runs in $O(m)$ time.

The actual monetary damage caused by the false positives might not be linear in the number of false positives, but instead could be some other (possibly arbitrary) function specified by the service provider. In this case, however, the algorithm will still produce correct results, and the cost function itself can be incorporated into the set of stopping rules τ, as we explain in section 4.5.

4.3 Algorithms for producing a permutation

There are several well-known methods for computing random permutations. Any method that has the following properties should be suitable for our approach:

– The permutation can be specified by a seed, i.e., given a seed value, the permutation could be reproduced from it. Recall that the set of storable seeds does not "access" all possible permutations of n elements, but only a random subset of $O(n^\ell)$ of these permutations[3]. This turns out to be enough in practical situations (see discussion in section 4.5).
– The algorithm allows concurrent computing of a mapping for a single element. It is then not necessary to compute the permutation mappings for $O(n)$ documents of the data collection at the access verification time just to obtain one of them that we are interested in. We can also directly compute the mappings for the m documents included in the order during card creation time without having to generate all of the n mappings.

As one example of a permutation satisfying there requirements, consider the case when $n' = n + 1$ is prime, g is a generator for that prime, and a permutation seed is specified as an integer x, $1 \leq x \leq n' - 1$. The permutation consists of any integer i, $1 \leq i \leq n' - 1$, mapping into $\pi_x(i) = x \cdot g^i \bmod n'$. It can easily be seen that the mapping π so defined is a permutation (i.e., there are no collisions). Of course, the use of x as a seed means that only n of the possible permutations of the n elements are "accessible". To extend the reachability of the seed from only n permutations to the full n^ℓ allowed by the available $O(\ell \log n)$ bits of storage, we would simply store as a seed ℓ distinct (rather than a single) such x values $x_1, \ldots, x_\ell$. Each x_j defines a permutation π_{x_j} in the manner described above: for the jth such permutation, i maps into $\pi_{x_j}(i) = x_j \cdot g^i \bmod n'$. The entire permutation described by this seed of length $\ell \log n$ bits is then the functional composition of the permutations $\pi_{x_1}, \pi_{x_2}, \ldots, \pi_{x_\ell}$ (in that order). There are n^ℓ possible choices for this permutation, as required.

In fact, any encryption function whose range and domain are [1, n], and whose key space is [1, n^ℓ], could be used for our purpose of permuting. That is, if x is the seed, then $\pi_x(i)$ is simply the encryption of i using x as key. The fact that n is too small for cryptographic security is not an issue here, because we are using encryption not to hide but rather to permute.

[3] In cases where a sequence of random numbers is needed by the permutation algorithm, the seed can be used to initialize a pseudo-random number generator.

4.4 Card operation

The algorithms presented above describe card generation, but they imply a corresponding operational use of the card, which we sketch here. We assume that the card is tamper-resistant, so that the unforgeability constraint is satisfied; techniques for achieving tamper-resistance can be found in the literature and are beyond the scope of this paper. Also, the card must anonymously authenticate itself to the server using a low-computation authentication suitable for smartcards. Policy enforcement is performed using the policy encoding placed on a card as follows. Given a document index i access to which is being requested from the server, and a card that stores a permutation seed s and ℓ intervals, the verification process takes the following steps:

– The card computes a permuted value of i as $p_i = \pi_s(i)$.
– The card searches its ℓ intervals for p_i to determine whether p_i is covered by one of them. Since we can sort all intervals before storing them on the card, this step can be done in $O(\log \ell)$ time using binary search.
– If p_i is covered by one of the ℓ intervals, the card requests the document i from the server. Otherwise, it notifies the user about access denial.

One can see from the above that the untraceability and unlinkability constraints of our design (goals of section 3.3) are satisfied: Each card anonymously authenticates itself and does not send any information to the server that might happen to be unique and used to link two transactions together. The card also does not require any additional sources of information to enforce proper access control and uses an efficient method for such enforcement, as required.

4.5 Economic analysis

This section analyzes the practicality of the scheme described above. We explore the possibility of using the scheme under different settings, and examine what policies a service provider might specify in order to use the model as efficiently as possible. We also make the “stopping criteria” that govern permutation selection process more precise.

4.5.1 Values of interest As input, we are given the size of the data repository n and the number of documents in a customer order m[4]. Other parameters of use for determining what an acceptable cost is are:

[4] In reality, we have the entire order $\{i_1, \ldots, i_m\}$ as an input parameter. For simplicity of presentation, we assume that the cost of each document is the same, and m can be treated as a sufficient representation of the set. Similar analysis can be carried out when document prices differ from one to another. Then each derived value that takes m as a parameter can be computed as a function of the set $\{i_i, \ldots, i_m\}$ itself.

$c_{card}(m)$ – the price a customer pays for an order of m documents, which can be a possibly arbitrary function of the documents that comprise the order.

$t(m)$ – the maximum number of requests to documents access to which was denied. Each card can count the number of attempts to view documents that were denied. When a customer requests a document not bound to the card, not only is the access denied, but also the permitted limit of unsuccessful requests is decremented. After t such attempts, the count reaches zero and the card is self-invalidated (i.e., the policy here is "t strikes and you are out"). This is to prevent customers from probing their cards for false positives, e.g., by trying all documents in the data repository. With this mechanism in place, each customer should be informed about t at the time of purchasing the card and should be given an explicit list of the documents included in his order.

$m'(n, m)$ – the number of documents that come for free with a card (i.e., the "false positives"). This value is computed as a by-product of Algorithm 2, and implicitly reflects the card's capacity ℓ.

$n'(n, m)$ – the number of documents in which an attacker is interested (other than the m he ordered). This value is useful in measuring the attacker's economic gain in case of discovering free accesses to documents. In the worst case, any free document can be valuable to the attacker. In the best case, the attacker has zero interest in anything outside the m documents she ordered.

4.5.2 Policy alternatives Each service provider deploying this approach might have one or more varying criteria that define an acceptable "false positives" cost of a card. Below we list policies that can be used during card generation to govern execution of Algorithm 1:

1. **Threshold for the number of false positives** m' that a card contains. This policy might dictate that the absolute value of the number itself is constrained (e.g., $f(m') \leq m'_{max}$), or its ratio to the number of documents in the repository or to the number of documents in the order is constrained by some threshold (e.g., $f\left(\frac{g(m')}{h(n)}\right) \leq m'_{max}$ or $f\left(\frac{g(m')}{h(m)}\right) \leq m'_{max}$, where $f(x)$, $g(x)$ and $h(x)$ are arbitrary functions of argument x). We may consider a policy that lists several conditions but requires satisfying a subset of them.
2. **Constraints on the gain from cheating**. In this type of policies, we perform analysis of cheating in terms of the attacker's loss vs. his gain after attempting to access t' out of the $n - m$ documents not included in his order. Suppose that $t' > t$. The expected gain from the attack in this case is the difference between the cost of the documents acquired for free from the list of n' documents of interest, and the cost of losing the card due to this behavior. The gain is then computed as the probability of successfully getting a free access to a document multiplied by the

document cost, while the loss is computed as the probability of losing the card multiplied by the cost of the card:

$$E(gain) \simeq t' \cdot \frac{c(m')}{n-m} \cdot \frac{n'}{n-m} - c_{card} \cdot Q \simeq \frac{t' c(m') n'}{(n-m)^2} - c_{card} \sum_{t''=t}^{t'} \binom{t'}{t''} q^{t''} p^{t'-t''}$$

where $c(m')$ is the cost of having access to m' documents computed according to some pricing function. Here $p = \frac{m'}{n-m}$ specifies the probability of not being caught, while $q = 1 - p$ is the probability of begin caught. Similarly, we can compute the expected gain when the number of unauthorized attempts is kept below the maximum, i.e., $t' \leq t$. In this case, the expected gain is computed based on the probability of getting free access, and there is no loss for the attacker:

$$E(gain) \simeq t' \cdot \frac{c(m')}{n-m} \cdot \frac{n'}{n-m} \tag{1}$$

In the worst-case scenario, the attacker might be interested in and benefit from any document acquired for free, i.e., $n' = n - m$, and we can also assume that $t' \simeq t$, to maximize the gain. Then equation (1) becomes:

$$E(gain) \simeq t \cdot \frac{c(m')}{n-m}$$

To keep the attacker's gain low, we might constrain this value by some threshold. Equation (2) gives such a constraint where the coefficient α plays the role of a threshold value that keeps the card's loss within a specified bound.

$$\frac{t \cdot c(m')}{n-m} \leq \alpha \cdot c_{card} \tag{2}$$

3. **Constraint on certain items being among the false positives**. The previous constraints take into account only the total number of false positives without distinguishing them, and do not account for the fact that the repository might contain a number of generally popular items. Thus, another constraint might be to lower the value of false positives for a customer by excluding such valuable items from the false positives. We refer to such items in high demand as "hot" items, and for each customer they can be either system-wide (the same for everyone), card-specific (based on the subscription order at card-creation time), or both.
Note that, from the privacy point of view, it is acceptable for the data owner to determine the hot items for a card based on the card's subscription order (which must be given anyway at the time of purchase, e.g., during anonymous card purchase at a vending machine or bookstore). Later on, as the card is used, the card does not disclose data about the subscription order or the card-specific forbidden hot items.
Once the set of such hot items for a subscription order is determined, the policy might state that there is a threshold on the number of such

items that can be among the false positives (the threshold can be stated similar to the total number of false positives in the first rule). Thus, if a particular instance of a card does not satisfy this requirement, a new instance should be generated.

4. **Timeout**. Under some policies, the card creation process might have to be carried within a certain period of time. Then if no suitable permutation is found during that interval, the best permutation tried so far is used.

Based on the policies listed above, we create a set of stopping criteria by possibly combining two or more conditions in such a way that what the card produced always satisfies the card issuer.

4.5.3 Sample policy Suppose a service provider employs a policy in which the number of attempts to access a document not included into the customer's policy configuration, t, cannot exceed 10% of the number of documents m in the customer's order. (Recollect that each customer at the time of purchase is given a list of all documents included in the order, so that t can be kept small.) The service provider also requires that the maximal customer gain from "false positive" documents cannot exceed 5% of the cost of the order. Evaluation parameters for a document permutation then can look like: $t = 0.1m$, $n' = n - m$, and $\alpha = 0.05$. Given n and an order consisting of m documents, we use Algorithm 1 to compute m'. According to equation (2), m' should satisfy the following condition:

$$\frac{0.1 \cdot m \cdot c(m')}{n - m} \leq 0.05 \cdot c_{card}$$

If the condition is not satisfied, the algorithm is invoked to try a new permutation.

With this policy in place, a card can be generated very efficiently for any order because the number of false positives is not required to be low. For instance, suppose that $c(m') \simeq m' \cdot c_1$ and $c_{card}(m) \simeq m \cdot c_1$, where c_1 is a unit price of a document. Then, in order to comply with the policy, we must have that $m' \leq \frac{n-m}{2}$, which is large and not difficult to achieve for any order of m documents. This tells us that the scheme can accommodate a wide range of reasonable policies.

4.6 Analysis of the approach

Our proposed solution is compliant with the desired design properties and minimizes the total number of false positives bound to a card. More precisely, the design of our scheme ensures that goals of transaction unlinkability and untraceability, unique policy representation, no additional sources of information, and fast access verification time listed in section 3.3 are met. The goal of forward compatibility is achieved by using unique policy representations that "capture" the state of the repository at the time of card generation and are self-contained. As we add more documents to the repository, the old cards can still be used, for instance, to reproduce permutations of the documents from the previous state of the repository and provide access to the documents from customer subscriptions.

Our permutation approach also guarantees a low rate of false positives, especially if this constraint is a part of the algorithm's termination criteria. Depending on the policies enforced by the service provider, the scheme can be evaluated on its time requirements, i.e., how long, on average, it might take to generate a card. Thus, it might or might not comply with the goal of fast card generation. If the service provider employs a policy that includes a timeout, then in-house card generation is always achievable. If, on the other hand, he places more weight on minimizing the number of false positives, then this constraint might be relaxed.

4.7 Structured data – trees

This section extends our approach to structured data collections such as trees. In many data repositories documents are stored in hierarchies, which makes it possible to utilize the repository's structure and reduce the number of false positives in the solution computed. Since in reality many customers do not select a random set of documents, but rather are interested in certain topics (which will guide their selection of items to be included in the order), great space savings (and thus a significantly reduced rate of false positives) can be achieved if instead of storing individual items we permit storing categories of items. In fact, the software that aids the user in selecting items to be included in his subscription can help to achieve space savings in cases of hierarchically structured documents. That is, it will provide the user with in option of selecting the entire section or category of documents at every level within the hierarchy, in addition to allowing selecting documents one by one. The quantification of such savings, however, cannot be performed in the general case, because it heavily depends on the type of the hierarchy and user patterns in selecting documents.

Now we present our approach for building user cards in case of hierarchically structured repositories. Suppose we are given a tree of n documents and a subscription order of m documents. The card's capacity is still $k = O(\ell \log n)$ bits or $O(\ell)$ records, but in this case each record, in addition to two numbers that specify a range, might contain some other information. We consider both positive and negative ranges for encoding documents on a card. We also consider two different types of placements: when a positive or negative assignment is placed on a node v, it can either affect the entire subtree rooted at v – we denote this case as *recursive* – or affect only the node on which the assignment is placed – we denote this assignment as *local.* The case where a depth parameter can be stored at v, so as to limit the depth of the subtree included, will be considered later in this section (such a depth parameter limits the depth of the nodes influenced by that range, so that nodes that are farther than that depth below v are not affected). When two ranges overlap, the more specific range (i.e., lower in the tree) is used. As before, the word "cost" is used as the "cost of the false positives" (not the dollar cost paid by the customer).

Throughout our algorithm, we use the following notations. For each node v, a cost of the subtree rooted at v can be computed in two different contexts: positive and negative. If a node v is evaluated in the positive context – the cost is denoted by $C^+(v)$, – this means that a positive range has been specified at its parent or above the parent in the tree. In this case, if no new range is placed at v or below, the entire subtree will be included in the final solution. In this context, only negative ranges placed at v or below have effect. Similarly, if a node v is evaluated in the negative context – the cost is denoted by $C^-(v)$, – then it means that a negative range has been specified at its parent or above, and by default the entire subtree will be excluded from the solution. If no context has been specified, we start in the negative context and assume that no nodes are included in the solution unless explicitly specified.

Our solution uses dynamic programming techniques; and as with any dynamic programming approach, the cost of an optimal solution at any given node v needs to be calculated for several cases that differ in the number of encoding slots available. Thus, we use $C^+(v,\ j)$ and $C^-(v,\ j)$ to mean the cost of encoding the tree rooted at v in positive and negative contexts, respectively, with j storage slots available, where $0 \le j \le \ell$.

Here we provide an algorithm for binary trees, which can naturally be extended to work for more general t-ary trees with $t \ge 2$. When working with binary trees, we typically use nodes u and w as child nodes of v. In order to compute a cost of a subtree rooted at node v, we need to consider two cases: computation of $C^+(v,\ j)$ and $C^-(v,\ j)$, which we describe subsequently. Let us consider non-leaf nodes first and then proceed with leaves of the tree. Time complexity of the algorithm for both binary and arbitrary t-ary trees is given later in this section.

4.7.1 Non-leaf nodes

Case of $C^+(v,\ j)$: When the cost is computed in the positive context, we need to consider three different cases.

Case 1: No record is placed at v. Then $C^+(v,\ j)$ is computed as:

$$C^+(v,\ j) = \min\{C^+(u,\ i) + C^+(w, j-i) + c_1 |\ 0 \le i \le j\} \quad ,$$

where c_1 is 1 if v is not in the order, and 0 otherwise.

Case 2: A negative recursive record is placed at v. This case cannot happen if v is included in the order. We compute the value as:

$$C^+(v,\ j) = \min\{C^-(u,\ i) + C^-(w, j-i-1) |\ 0 \le i \le j-1\}$$

Case 3: A negative local record is placed at v. This case also cannot happen if v is included in the order. To compute $C^+(v,\ j)$, we use:

$$C^+(v,\ j) = \min\{C^+(u,\ i) + C^+(w, j-i-1) |\ 0 \le i \le j-1\}$$

After computing all of the values above, $C^+(v,\ j)$ is assigned the minimum of the three values.

Case of $C^-(v,\ j)$: For the negative context there are also three possible cases.

Case 1: No record is placed at v. This case cannot happen if v is included in the order. The formula for computing $C^-(v,\ j)$ is as follows:

$$C^-(v,\ j) = \min\{C^-(u,\ i) + C^-(w, j-i) |\ 0 \leq i \leq j\}$$

Case 2: A positive recursive record is placed at v. In the formula below, c_1 is set to 1 if v was not included in the order, and it is 0 otherwise:

$$C^-(v,\ j) = \min\{C^+(u,\ i) + C^+(w, j-i-1) + c_1 |\ 0 \leq i \leq j\}$$

Case 3: A positive local record is placed at v. This case normally does not happen when v is not in the order. To compute $C^-(v,\ j)$, we use:

$$C^-(v,\ j) = \min\{C^+(u,\ i) + C^+(w, j-i-1) + c_1 | 0 \leq i \leq j\}$$

Analogously to the previous case, $C^-(v,\ j)$ receives the value of the minimum of the three values computed in these cases.

4.7.2 Leaf nodes

Case of $C^+(v,\ j)$: If $j > 0$ and v is not in the order, then we can exclude the node from the solution by placing a negative record at it. In this case, the cost $C^+(v,\ j)$ is 0. Otherwise, no record can be placed at the node; and the cost $C^+(v,\ j)$ is 0 if v is included in the order, and 1 otherwise.

Case of $C^-(v,\ j)$: If $j = 0$ and v is included in the order, then $C^-(v,\ j)$ should be set to $+\infty$ to prevent this configuration from being chosen, as it does not satisfy the algorithm's requirements. In all other cases, $C^-(v,\ j)$ is 0.

4.7.3 Complexity analysis To compute the cost of an order, we use the above rules to compute $C^-(root, \ell)$. Every documents i included in the order is taken into account at the time of computing the cost of the subtree rooted at node i. For a tree of n documents and card's capacity of ℓ slots, this algorithm runs in $O(n \cdot \ell^2)$ time for binary trees. For arbitrary t-ary trees the algorithm gives $O(n \cdot \ell^t)$ time.

4.7.4 Access verification Once a solution using the above techniques has been computed and stored on the card, the card will perform access verification as follows:

1. On user request to access item i, the card first checks whether i is stored on the card in positive context. If true, return accept. If i is stored on the card in negative context, return reject. Otherwise, proceed with the rest of the algorithm.
2. The card sends a request to the server to retrieve the nodes on the path from i to the root of the tree.
3. The server replies with $\mathcal{P} = \{p_1, \ldots, p_d\}$, which is the nodes on the path ordered from i to the root of the tree.

4. For $i = 2$ to d, the card performs: if p_i is stored on the card with a recursive label, then if the context is positive return accept, and if the context is negative return reject. If p_i is not on the card or is stored with a local label, proceed with the next node p_{i+1}.
5. If none of the p_i's was found on the card, return reject.

If $\mathcal{P}$ is short enough to fit on the card, then the server can send it all at once. Otherwise it sends the path in batches small enough to fit on the card. In practice, $\mathcal{P}$ is likely to be small, as typical hierarchies tend to have small height. For instance, hierarchies such as collections of newspaper articles might have a very large total number of nodes, but the depth of the hierarchy will be limited by a small constant.

4.7.5 An extension to records of variable depth Let h be the height of the tree. The above dynamic programming approach can be extended to include all possible heights for each node v. This means that when we compute a cost of a subtree $C^+(v,\ j)$ or $C^-(v,\ j)$, we now can specify the depth of the record placed at v, which can vary from 1 to the height of the subtree rooted at v. In this case, there is no need to distinguish between local and recursive nodes any more, as they are replaced by a single record in which the desired depth is specified. We do not include the algorithm's details because they can easily be derived from the previous algorithm.

For a t-ary tree, this modification implies a factor of h (but not h^t) in the algorithm's time complexity, because any record placed at the parent covers one child's subtree at same depth as for another child's subtree. Thus, it takes h times as long to compute the cost of each subtree.

4.7.6 Randomization The above tree algorithm was static, and the solution generated would always be the same for the same set of documents. Thus, it does not satisfy the desirable requirement that even identical subscriptions have different solutions. This can be remedied by using for each subscription a random re-naming π of the node names and then using the new names to describe the subscription on the card. More precisely, at the card generation time, we first run the dynamic programming algorithm to generate card representation. Next, we choose a random permutation π over the n nodes of the hierarchy (the permutation must satisfy the properties listed in section 4.3) and apply the permutation to the nodes used in the card representation. We then store such permuted representation of access rights on the card.

When the user requests access to a document i, the card, as before, first asks the server for the i-to-root path $\mathcal{P}$ in the hierarchy. Then it applies the random re-naming π to all of the nodes of that path $\mathcal{P}$, and it finally compares those values with its stored subscription description to determine whether access should be granted.

4.8 Dynamic changes to the repository

In this section we first focus on handling dynamic changes to the repository consisting of unstructured data (i.e., a mere collection of numbered items), and later address this issue for hierarchically structured collections of documents (i.e., trees).

As was shown in section 4.6, the scheme is forward compatible when the documents within the repository are not organized into a structure, i.e., addition of new items to the repository is modeled as assigning to each of them a number strictly larger than the current number of documents in the repository. For instance, if the current data repository has n documents (numbered 1 through n) and there are cards that generate permutations over these n documents, addition of new d documents will result in their numbers being $n+1, \ldots, n+d$, and a newly generated card will store permutations over these $n+d$ documents.

Deletions of documents are also possible using our scheme. First, we would like to point out that a document will not be taken out from the repository while that document is present in at least one current customer subscription (information about documents in use can be collected at the time of subscription purchase; note that this does not violate privacy of the customers). When an unpopular is, however, being removed from the repository, we do the following: the document is simply removed without affecting the rest of the repository (i.e., the remaining documents are not re-numbered). This leaves a "hole" in the consecutive numbering of documents, but if the next new document is assigned this number, the effect of discontinuous numbering is mitigated. Since data repositories tend to grow (but not shrink) over time, removal of documents will not result in unnecessary increase of the number of items being permuted.

Now we would like to discuss handing of dynamic changes in hierarchically structured data repositories. Unlike in the unstructured case, now in addition to insertions and deletions, we also need to deal with restructuring of the items in the repository.

In general, handling dynamic changes to the hierarchy requires more careful treatment, because the structure "captured" by the access right representations on existing cards must be preserved. Therefore, in order to ensure compatibility, any changes we introduce cannot modify the structure of the previous state of the data repository. That is, new branches and subtrees can be added at the root of the tree; and documents to which no user is currently subscribed can safely be removed from the tree (without removing from the data structure the parts of the tree that correspond to such documents).

Structural changes that affect the old parts of the tree can be done using scheduled periodic updates. That is, insertions are introduced as they occur, but such documents are attached as new branches at the root level; deletions and re-structuring, on the other hand, are delayed until the next update. During an update, the structure of the tree is modified using all pending deletions and re-structuring (parts added to the tree after the last update are also moved to their appropriate places, if necessary). All users submit their subscription preferences and are issued new cards. (Note that in our model it is impossible to recover exact access rights that a user had using her access right representation, therefore we allow each user to select possibly a different set of documents, the total value of which corresponds to the subscription price paid.)

5 Approach Based on Minimal Perfect Hash Functions

In addition to the notation described in section 3.2, in this section we use the following notation: A hash function $h : X \rightarrow Y$ is called a *perfect hash function* if it is 1–1, i.e. $\forall x_1, x_2 \in X$, $h(x_1) \neq h(x_2)$ iff $x_1 \neq x_2$. In other words, perfect hash functions never result in collisions. A hash function $h : X \rightarrow Y$ is called a *minimal perfect hash function* (MPHF) if it is 1–1 and for which $|X| = |Y|$. An *order-preserving* MPHF (OPMPHF) also has the property that it maps the ith smallest element of X into the integer i.

In what follows we use f to denote a minimal perfect hash function that maps $\{i_1, \ldots, i_m\}$ to $\{1, \ldots, m\}$ without collisions. Also, functions f', f'' denote order-preserving MPHFs each of which maps $\{i_1, \ldots, i_m\}$ to $\{1, \ldots, m\}$ without collisions and in an order-preserving manner (i.e., $f'(i_j) = j$).

In this section we first give a preliminary solution that utilizes minimal perfect hash functions and is described in section 5.1. For a subscription order of m documents, it results in access rights representations $O(cm)$ space and the probability of false positives being 2^{-c}, where c is an adjustable parameter. The second, improved, solution uses order-preserving minimal perfect hash functions to achieve significantly better asymptotic performance: with $O(cm)$ storage space available, the probability of false positives is m^{-c}. It can be found in section 5.2. Section 5.3 describes extensions to the schemes: it discusses how certain items can be completely eliminated from the possible false positives and also covers space utilization techniques for hierarchies. Finally, section 5.4 addresses dynamic changes to the repository.

5.1 A preliminary solution

Given a card that can store $k = O(cm)$ bits, this approach gives us: (i) a card creation time polynomial in m, and (ii) the probability of false positives 2^{-c}. Note that it is reasonable to assume that cards can store cm bits. The reason is that this space will be small for relatively small orders; for larger, more expensive orders one can use cards of larger capacity, the manufacturing cost of which can be offset by the amount charged for the subscription order.

In what follows, H is a keyed cryptographic one-way hash, whose key is unique to each card (to make false-positive information sharing impossible);

the key's purpose is not cryptographic security, but rather making each card unique. The k bits available do not include the bits needed for storing the key for the hash H, which would be small in practice. For instance, a 20-bit key would result in a million different cards that can request identical m items yet be different; and the possibility of such sharing when the two cards correspond to different sets of documents significantly decreases. An alternative to a keyed H is to have the same hash function H for all cards, but force the random choices made during the computation of a suitable minimal perfect hash function to vary from card to card.

5.1.1 Card creation

1. Compute a minimal perfect hash function f for $\{i_1, \ldots, i_m\}$. Store f in the card using $O(m)$ bits (according to [22], a MPHF for m random strings can be stored using bm bits, where b is a constant and can normally be 2). This leaves $k' = O(cm)$ bits available for what follows.
2. Partition these k' bits into m blocks of c bits each; call them blocks B_i ($i = 1, \ldots, m$).
3. Let the hash function $H(x)$ produce a c bits long hash of x (e.g., by considering only c of the 160 bits it produces in case of SHA-1). For every item $i \in \{i_1, \ldots, i_m\}$, if $f(i) = j$, then set the bits of block B_j on the card equal to $H(i)$.

5.1.2 Access verification Every time a customer uses her card to request access to an item i, the card performs the following:

1. Compute $f(i)$; assume $f(i) = j$.
2. Compare the c bits of the card's block B_j to the corresponding computed c bits of $H(i)$.
3. Access is allowed if these c bits match, and denied otherwise.

5.1.3 Analysis

Theorem 1 *Given $k = O(cm)$ storage space, the above MPHF-based approach produces in time polynomial in m a solution with the properties of (a) transaction unlinkability and untraceability, (b) unique policy representation, (c) no additional sources of information, (d) forward compatibility, and (e) probability of false positives 2^{-c}.*

Proof Card creation takes time polynomial in m because a MPHF can be generated in polynomial time [23]. Given $k = (c + b)m = O(cm)$ space, the probability of a false positive is less than 2^{-c} (see, e.g., [22] for more detail).

Transaction untraceability is achieved because the card anonymously authenticates to the server and then everything else it sends is a request for a specific data item with no personal or card-specific information. By the same argument, any two transactions are also unlinkable.

Unique policy representation is achieved through the use of the keyed hash function H or, alternatively, by randomizing f itself. Each card will also stay operational as we add more items to the data repository because the card is dependent on the purchased items and contains no information about other items or the size of the data repository. This means that the forward compatibility requirement is satisfied. Finally, by design this scheme does not use any additional sources of information. □

5.1.4 Case where $c = \log m$ If in the above $c = c' \log m$ where c' is constant, then the scheme has $k = O(c'm \log m)$ bits of storage and an $m^{-c'}$ probability of false positive. In such a case, however, the following simpler scheme that achieves the same bounds can be used. We use a keyed hash function F (not a perfect one — collisions can occur) that maps items in the range $[1, n]$ into $[1, m^{c'+1}]$. An example of such a function that we use in our further discussion is $F(i) = H(i) \bmod m^{c'+1}$, where H is a keyed cryptographic one-way hash function. What the card stores is the (at most m) elements of $[1, m^{c'+1}]$ to which the subscription items map. It allows access to a requested item i iff $F(i)$ is stored on the card. Since each of the (at most) m numbers stored is $(1 + c') \log m$ bits long, the total space needed is $O(c'm \log m)$ bits. The probability of a false positive is no greater than $m/m^{c'+1} = m^{-c'}$. This matches the MPHF scheme's performance if $k = m \log m$, but it cannot be used if $k = o(m \log m)$. When it can be used, however, it has the potential for the following heuristic improvement in its space usage: The m stored elements from $[1, m^{c'+1}]$ could be such that the trie implied by their bit representations makes further space savings possible (by storing common prefixes or other common bit patterns only once). The expected space needed to store the trie, however, remains $O(m \log m)$ bits so the savings are by no more than a constant factor, and the multiplicative factor of $\log m$ in the space complexity remains.

The scheme in the next section achieves the same false positives probability performance of m^{-c} but without the multiplicative factor of $\log m$ in the space used.

5.2 An asymptotically better solution

Given $k = O(cm)$ space available on the card, the approach described in this section and which is based on usage of order-preserving MPHFs gives us: (i) a card creation time polynomial (in fact, linear) in m, and (ii) the probability of false positives m^{-c}, where c is an integer parameter that can be chosen so as to achieve a desired PFP. For large enough m (which we assume is the case since $m > k/\log n$), however, it is sufficient to have $c = 1$. We start with describing the $c = 1$ version of the scheme, after which we extend it to larger values of c.

5.2.1 Card creation As usual, we deal with a subscription order $\{i_1, \ldots, i_m\}$, where $i_1 < i_2 < \ldots < i_m$. We use two order-preserving minimal perfect hash functions f' and f'', each computed for this subscription order: $f'(i_j) = j$

and $f''(i_j) = j$ for all $j \in [1, m]$. To see why we use different functions f' and f'', we first need to recall that the construction of an order-preserving minimal perfect hash function involves many random choices along the way, and f' and f'' will differ through those different random choices. While the effect of such functions on the elements of the set $\{i_1, \ldots, i_m\}$ is fixed and well known for all i_j (i.e., $f'(i_j) = f''(i_j) = j$), their effect on elements not in the set $\{i_1, \ldots, i_m\}$ is arbitrary. Consequently, we use two different functions f' and f'' for their different effects on randoms r *that are not in set* $\{i_1, \ldots, i_m\}$. This is an unusual use of such functions because we use their *random effect* on an r that is *not* in the set, as much as their predictable effect on an i_j from the set. The card hence stores f' and f'', which take $O(m)$ bits of space.

While the effect of those random choices on a random $r \notin \{i_1, \ldots, i_m\}$ has not been investigated in the literature, we postulate that the existing OPMPHF schemes can be used to hash such an r uniformly on the interval $[1, m]$. That is, each of $f'(r)$ and $f''(r)$ is random and uniformly distributed over $[1, m]$. What follows is subject to this assumption[5].

5.2.2 Access verification To verify a request to access an item i, the card needs to perform the following steps:

1. It computes $f'(i)$ and $f''(i)$.
2. Access is granted if $f'(i) = f''(i)$, and denied otherwise.

5.2.3 Extension to higher values of c The above description results in the PFP being $\frac{1}{m}$. To obtain versions of the scheme with PFP of $\frac{1}{m^c}$ for $c > 1$, instead of using two functions f' and f'', we use $c+1$ such functions: access to i is granted if all $c + 1$ functions map i into the same value, and is denied otherwise. Of course, different random parameters are selected when constructing each of these $c+1$ functions, and the space complexity becomes $O(cm)$ bits.

Note that if the value of c is relatively large compared to m, it might be difficult or even impossible to generate $c+1$ different functions for those m items. In such a case, either the value of c might be lowered, or the space needed to store these $c + 1$ functions might have to be increased (each function will still require $O(m)$ bits but with a larger than optimal constant).

[5] It is possible that OPMPHF representations that use an optimally small number of bits bm will not involve many random choices for certain elements of the set $\{i_1, \ldots, i_m\}$. This means that $f'(r)$ and $f''(r)$ might not be truly independent for some values of r. To magnify the randomization effect of the functions on such r's, we might want to increase the space occupied by the functions by increasing the value of the constant b, and make sure that the number of random choices during function generation is large. Obviously, this topic deserves further investigation and formal treatment.

5.2.4 Analysis

Theorem 2 *Given $k = O(cm)$ space, the above OPMPHF-based approach produces in time polynomial in m a solution with the properties of (a) transaction unlinkability and untraceability, (b) unique policy representation, (c) no additional sources of information, (d) forward compatibility, and (e) probability of false positives m^{-c}.*

Proof Each of the $c + 1$ functions can be computed in linear time and space [21], therefore the claimed card creation time holds. We now argue that the probability of a false positive is m^{-c}. First we note that, for an $r \notin \{i_1, \ldots, i_m\}$ to be a false positive, all of the $c + 1$ functions must map r into the same value. Recall from our above assumption that the choices of different random parameters for each such function f' randomize $f'(r)$ uniformly over $[1, m]$, and thus the probability that, given a random r, $f'(r)$ will fall into a specific cell is $\frac{1}{m}$. By choosing the different functions' random parameters independently, this effectively makes the value of $f'(r)$ independent of the other $f''(r)$ values. The probability that the $c+1$ functions map r into the same value is therefore $1 \cdot \left(\frac{1}{m}\right)^c = m^{-c}$.

Property (b) is ensured through the random choices in selection of the functions, and properties (a), (c), and (d) are by the same arguments as in the proof of Theorem 1. □

5.3 Extensions

In this section we provide two extensions to the scheme described. Namely, we show how to completely eliminate certain items from the false positives and also how to extend our scheme to hierarchical data structures.

5.3.1 Decreasing the value of false positives Recall that in section 4.5.2 we talked about excluding certain items of high demand from the list of false positives. Performance of any of the MPHF-based schemes can also be improved with respect to the cost of false positives if we can ensure that such generally popular (so-called "hot") items are not among the false positives. Also recall that for each customer these items can be system-wide, card-specific, or both.

There are various ways of ensuring that such hot items are not among the false positives of a customer order. In what follows we describe different ways of achieving it. Before proceeding with the specific approaches, we first describe the pre-processing step that applies to all of them. The pre-processing requires the algorithm to determine the following three sets of documents: (i) the set of documents in the customer order; (ii) the set of system-wide hot items; and (iii) the set of card-specific hot items. Note that the set (ii) or (iii) could be empty, depending on the application, the repository, and document characteristics. Next, we combine the sets (ii) and (iii) into a single one and call it a "negative subscription list." Let m_h

denote the cardinality of such a set. Now we are ready to proceed with the specific techniques.

1. One way of isolating such hot items from the rest of the documents is by incorporating their isolation into the random choices used in card generation. That is, after we make random choices at the card-creation time, we can evaluate the resulting encoding against the list of the hot items. If, as a result, any of them (or any number of them above a certain threshold) is among the false positives, we repeat the card-creation process with another set of random choices until the desired level of false positives with respect to these hot items is achieved. Note that the time needed to generate a card with none of the hot items among the false positives will be directly proportional to the number of such hot items. That is, given truly random choices and, for instance, having $k = O(cm)$ and $PFP = 2^{-c}$ (for the MPHF-based approach), the probability that none of the hot items are among the false positives is $\left(1 - \frac{1}{2^c}\right)^{m_h}$. Thus, the expected number of times one needs to invoke the card-creation algorithm is $(1 - \frac{1}{2^c})^{-m_h} - 1$.
2. Another approach is to map the popular items to a region different from the legitimate subscription items on the card. More precisely, given the set of subscription items and the negative subscription list, we can use the techniques used in card generation on both of them, with the difference that access to the items in the first set is *permitted*, while access to the items from the second set will be *denied*. First, we use our techniques to generate a representation of access rights for the subscription order as before. Then we test every item on the hot list against this representation and determine which of them are among the false positives (their number will be $PFP \cdot m_h$ on average and we denote that number as m'_h). Next, we create a list consisting only of the hot items that happened to be false positives, and apply our techniques to that list mapping it to a separate region on the card. Access to every document that successfully hashes to that second region will be denied (for that reason we need to check all documents on the original subscription list to ensure that access to them will not be denied; if any of them do hash successfully on the denied region, re-run this algorithm as many times as needed until none of the subscription documents land on that region successfully).
   Once the card is created using the above description, it is issued to the customer. When the customer requests an item, the card first checks that access to it is permitted (using the first region) *and* then it checks that it is not among the items access to which should be denied (using the second region).
   The foregoing approach obviously increases the space necessary to represent a subscription order. Thus, now instead of requiring the card's storage space k be, for instance, $O(cm)$, it will be a function of m and m_h (e.g., $k = O(c_1 m) + O(c_2 m'_h)$). This increase in the space, however, may be worthwhile, if it costs less than the cost associated with the hot items being among the false positives. It is clear that this technique

should be used when the size of the hot items list is large enough, so that it is difficult to prevent these items to be among false positives using other techniques.

3. Finally, another way to deny access to hot items is to merge the list of the subscription items with the negative subscription list, but place a special mark on each of them indicating whether access to the document should be permitted or denied. Thus we run our regular card-creation algorithm on the joint list, but mark each resulting hash with a permit or a deny mark. When a customer requests an item (having the card generated according to this approach), access to that item will be permitted only if passes the card test *and* the mark on the corresponding hash is of the access type.
   This method results in even larger storage increase than in the previous case (i.e., $k = O(c_1 m) + O(c_2 m_h)$ instead of $k = O(cm)$), but it is simple and fast to produce. Thus, it can be useful in cases when the list of hot items is short, but the damage of having them among the false positives is large (i.e., we want absolutely none of them to be among the false positives).

Which method to choose will depend on the target application and the specific customer order. More precisely, the size of the subscription order and the number of hot items access to which must be denied will determine what approach to use. In general, it is possible to combine the above techniques into a hybrid scheme, where, for instance, access to a fraction of the hot items is prevented through the random choices (using a certain number of iterations as in the first technique) and access to the remaining fraction of the hot items is prevented through a negative list. We believe such a hybrid solution will be best in terms of time and space resources used.

In general, in our schemes the rate of false positives is very small (e.g., for $m = 100$ and $c = 3$ the PFP is one in a billion), and therefore even if the list of hot items is long, only a tiny fraction of them will be among the false positives, which needs to be isolated. This permits us to use the above special treatment for those few items, if we want absolutely no hot items to be among the false positives.

5.3.2 Improving space utilization for hierarchies As was mentioned in section 4.7, hierarchically structured data repositories provide additional possibilities for efficiently utilizing storage space on the cards and therefore minimizing the rate of false positives. For tree-like hierarchies, the objects in the repository can correspond to the leaves of the tree. In addition, the internal nodes correspond to categories of objects and are also marked with unique identifiers. Then great space savings can be achieved if now instead of storing all m items, we store nodes of the tree, the entire sub-trees of which are among the m items.

Our techniques can be extended to such hierarchical repositories if we assume that for every item i its “path to the root” can be obtained from the server. One possibility is to use the following procedure to obtain a list of items to be included in the representation of access rights. In what follows, assume that the hierarchy is a tree (of any degree), and the leaf nodes (that correspond to actual items) are numbered 1 through n (from left to right). Let us also assume that the total number of nodes in the tree is N; the internal nodes are numbered $n+1$ through N; and the subscription is given as a list of m leaf nodes $\{i_1, \ldots, i_m\}$ sorted in the increasing order. The resulting list of nodes (which will be used to represent access rights) is denoted $L = \{j_1, \ldots\}$. Note that for each $j_k \in L$, $1 \leq j_k \leq N$ and $|L| \leq m$. The following procedure creates such a list L:

1. Set $L = \{i_1\}$.
2. For each $k = 2, \ldots, m$, do:
   (a) Set $L = L \cup \{i_k\}$.
   (b) Let p be the parent of i_k and let $c_1, \ldots, c_p$ denote the children of p. If every $c_j \in L$, set $L = (L \setminus \{c_1, \ldots, c_p\}) \cup p$; and recursively repeat this step going up in the hierarchy (i.e., starting with the parent of p until no more changes are made to L.

Having the list L, we then apply the MPHF techniques to it to generate the card itself. Note that this will introduce space savings because now instead of having m records, we will need $|L| \leq m$ records to represent the access rights, in some cases having $|L| \ll m$. We would like to point out again that the exact savings will be determined by the type of the data structure and user patterns in selecting items for their subscriptions.

Note that unlike in the techniques given in section 4.7 for tree-like hierarchies, the representation of access rights obtained in the above (or similar) fashion cannot introduce any false positives: in order to meet the claimed rate of false positives, no false positives are permitted in addition to those introduced through the use of minimal perfect hash functions.

Adoption of the above techniques will affect card operation at the time a user makes a request to access an item. That is, now instead of directly applying the hash function to the document being requested (assume it is i) and checking for its access rights, the card will need to retrieve information about the nodes on the way from i to the root. Then the card checks every item on this list and assumes that the user has access rights to i if she has access to at least one node on the list.

It is clear that the foregoing techniques involve more interaction with the server (i.e., retrieval of public information stored at the server). It is not clear, however, how to avoid this extra interaction: while it is well known that ancestral relationships in a tree are completely described by two linear listings of its nodes (e.g., preorder and postorder), this is not immediately exploitable because of the randomization introduced by the hashes. This clearly deserves further investigation.

An additional complication is that, in hierarchically structured objects, care must be exercised to ensure that certain nodes are not among the false positives. For instance, during the card creation process we must ensure that the root of the tree and other nodes high in the hierarchy are not among the false positives. This is especially important now, when information about the hierarchy (and unique numbers associated with nodes higher up in the hierarchy) is publicly available. Techniques of the previous subsection then can be applied to this case to exclude the special items from the possible false positives.

5.4 Dynamic changes to the repository

Similar to the scheme given in section 4, we need to consider the behavior of this second approach with respect to dynamic changes to the repository, which should be done for both unstructured collections of documents and repositories in which the documents are organized in a hierarchy (more precisely, in a tree). In either case, this scheme exhibits the same characteristics with respect to dynamic changes as those of the first scheme. Thus, the results reported in section 4.8 for the first scheme apply to this case as well and are not repeated here.

6 Comparison of the Schemes

In this work we gave two schemes for minimizing space requirements to permit user access to items of their choice from a large data repository in a privacy-preserving manner. Both of the schemes comply with the design goals of: transaction untraceability and unlinkability, unique policy representation, single storage device, fast operation, and forward compatibility of the schemes. They, however, have drastic differences: while the first approach assumes a fixed amount of storage space available for a subscription order and attempts to produce an encoding that minimizes the number of false positives for that order, the second approach has explicit space requirements that depend on the order size, but it guarantees a low rate of false positives. Thus, the second method assures the bounds on PFP given the space available, but the first method is the best-effort approach that attempts to meet the feasibility criteria.

Properties of the schemes that are based on minimal perfect hash functions (i.e., from section 5) are summarized in Table 1. The reason why the table lists performance only of the schemes based on MPHFs is that the performance of the permutation-based approach will be completely determined by the stopping criteria used, does not have fixed upper bounds, and thus cannot be directly compared to other schemes.

Among all schemes given in section 5, the tree-based $k = O(m \log m)$ space approach described in section 5.1.4 is the simplest to implement, and its space usage can be heuristically lowered as described in that section. This approach, however, cannot be used if $k = o(m \log m)$, whereas the MPHF-based scheme can work in cases when $k = o(m \log m)$, e.g., when $k = O(m)$. In general, both of these approaches give the same rate of false positives

Scheme / Space	$k = O(cm)$	$k = O(cm \log m)$
MPHF-based	2^{-c}	m^{-c}
OPMPHF-based	m^{-c}	$m^{-c\log m}$

Table 1 The rates of false positives of the schemes for different storage space bounds. Here m is the number of items in the subscription, k is the storage space, and c is a constant.

of m^{-c} if $k = O(cm \log m)$. The OPMPHF-based approach of section 5.2 achieves the same false positives rate of m^{-c}, but with an asymptotically lower requirement for space: $O(cm)$ bits.

Let us next provide a heuristic to determine what scheme should be used for card generation, assuming that all of the approaches are available to build a user's card. Let $I_m = \{i_i, \ldots, i_m\}$, and let f be a function that given a set $I_m \subseteq \{1, \ldots, n\}$ determines its value and outputs the maximum allowable card size k. Note that, depending on the application and available technology, f might be independent of its argument and always output a fixed size; or it also might use its argument to select one of the few available card sizes. Let $C \in \{0, 1\}^k$ denote an instance of access rights representation for a set I_m. Also let us use the following naming conventions for card generation algorithms:

SingleBitScheme: Can be used when the card's capacity k exceeds the repository size n. Then the encoding on a card will allocate one bit per repository document and set the bit for each $i_j \in I_m$ to 1 and all other bits to 0. There are no false positives in this case.

ListItemsScheme: Can be used when the card's capacity k exceeds $m \log n$. Then the encoding on a card will list all items $i_j \in I_m$. Similar to the previous scheme, PFP$= 0$ in this case.

PermutationScheme: The scheme described in section 4. This algorithm requires a set of evaluation rules (or stopping criteria) τ for its execution. We also define p, $0 < p < 1$, to be the portion of the card dedicated to storing the seed of the random permutation.

MPHFScheme: The MPHF-based scheme described in section 5.1. In addition to the information about the subscription order and card capacity, the algorithm takes two other arguments c_1 and b_1. Here c_1 serves the role of the configurable parameter c in the scheme (i.e., the probability of false positives will be 2^{-c_1}), and b_1 is a constant such that $b_1 \cdot m$ space is needed to store the MPHF for m items. Note that c_1 may be a function of m or I_m.

OPMPHFScheme: This is the scheme described in section 5.2. Similar to the MPHFScheme($\cdot$), this algorithm takes parameters c_2 and b_2, where c_2 represents the desired PFP (i.e., PFP$= m^{-c_2}$ and c_2 is possibly a function of the subscription order) and $b_2 \cdot m$ bits are needed to store the OPMPHF for m items. Note that b_2 will be higher than b_1.

```
SelectScheme($n, I_m, p, \tau, c_1, c_2$):
  compute $k = f(I_m)$
  if $k \geq n$
    $C$ = SingleBitScheme($n, I_m, k$)
  else
    if $k \leq m \cdot \lceil \log_2 n \rceil$
      $C$ = ListItemsScheme($n, I_m, k$)
    else
      if $k \geq c_2 b_2 m$
        $C$ = OPMPHFScheme($n, I_m, k, c_2, b_2$)
      else
        if $k \geq (c_1 + b_1)m$
          $C$ = MPHFScheme($n, I_m, k, c_1, b_1$)
        else
          $C$ = PermutationScheme($n, I_m, k, p, \tau$)
return $C$
```

Fig. 2 Algorithm for selecting a scheme for card generation.

Figure 2 gives an algorithm for selecting the most suitable card generation scheme assuming that all of the above card creation algorithms are available. It assumes that the data repository is an unstructured collection of documents.

Finally, we would like to summarize that our solutions can be used for different applications, with the most intuitive ones being digital libraries that might contain books, articles, magazines, and also music, video, and other objects. With such systems in place, a customer can purchase a subscription to the items of interest from stores, or libraries, and have anonymous access to the documents from many convenient locations as well. Other usages include access to locally stored (encrypted) objects, where software trusted by the document owners mediates access (effectively playing the role of the server) and on-demand decrypts the objects that the user is authorized to access.

References

1. M. Atallah and J. Li. Enhanced smart-card based license management. In *IEEE International Conference on E-Commerce (CEC'03)*, pages 111–119, June 2003.
2. G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Advances in Cryptology – CRYPTO'00*, volume 1880 of *LNCS*, pages 255–270, 2000.
3. G. Ateniese and B. de Medeiros. Efficient group signatures without trapdoors. In *ASIACRYPT'03*, volume 2894 of *LNCS*, pages 246–268, 2003.
4. T. Aura and D. Gollmann. Software license management with smart cards. In *USENIX Workshop on Smart Card Technology*, May 1999.

5. E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. Working Paper, Sloan School of Management, MIT, 2002. `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=299935`.
6. E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, May 2001.
7. E. Bertino, S. Castano, and E. Ferrari. Securing XML documents with author-$\mathcal{X}$. *IEEE Internet Computing*, 5(3):21–31, 2001.
8. E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 5(3):290–331, August 2002.
9. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
10. D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *Advances in Cryptology – CRYPTO'04*, volume 3152 of *LNCS*, pages 41–55, 2004.
11. D. Boneh and M. Franklin. Anonymous authentication with subset queries. In *ACM Conference on Computer and Communication Security (CCS'99)*, pages 113–119, November 1999.
12. A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. Allerton Conference, 2002.
13. M. Bykova and M. Atallah. Succinct specifications of portable document access policies. In *ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, June 2004.
14. J. Camenisch and J. Groth. Group signatures: Better efficiency and new theoretical aspects. In *Conference on Security in Communication Networks (SCN'04)*, volume 3352 of *LNCS*, pages 120–133, 2005.
15. J. Camenisch and M. Michels. A group signature scheme with improved efficiency. In *Advances in Cryptology – ASIACRYPT'98*, volume 1514 of *LNCS*, pages 160–174, 1998.
16. D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.
17. Z. Czech, G. Havas, and B. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, October 1992.
18. D. Damiani, S. De Capitani Di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5(2):169–202, May 2002.
19. P. Devanbu, M. Gertz, A. Kwong, C. Martel, and G. Nuckolls. Flexible authentication of XML documents. In *ACM Conference on Computer and Communications Security (CCS'01)*, November 2001.
20. L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.
21. E. Fox, Q. Chen, A. Daoud, and L. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9(3):281–308, July 1991.
22. E. Fox, Q. Chen, and L. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Annual International ACM SIGIR*, pages 266–273, 1992.

23. E. Fox, L. Heath, Q. Chen, and A. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, January 1992.
24. J. Kilian and E. Petrank. Identity escrow. In *Advances in Cryptology – CRYPTO'98*, volume 1462 of *LNCS*, pages 169–185, August 1998.
25. J. Kim, S. Choi, K. Kim, and C. Boyd. Anonymous authentication protocol for dynamic groups with power-limited devices. In *Symposium on Cryptography and Information Security (SCIS'03)*, volume 1/2, pages 405–410, January 2003.
26. C. Lee, X. Deng, and H. Zhu. Design and security analysis of anonymous group identification protocols. In *Public Key Cryptography (PKC'02)*, volume 2274 of *LNCS*, pages 188–198, February 2002.
27. M. Mitzenmacher. Compressed bloom filters. In *ACM Symposium on Principles of Distributed Computing*, August 2001.
28. P. Persiano and I. Visconti. A secure and private system for subscription-based remote services. *ACM Transactions on Information and System Security*, 6(4):472–500, November 2003.
29. A. Santis, G. Cresenzo, and G. Persiano. Communication-efficient anonymous group identification. In *ACM Conference on Computer and Communication Security (CCS'98)*, pages 73–82, November 1998.
30. S. Schechter, T. Parnell, and A. Hartemink. Anonymous authentication of membership in dynamic groups. In *Financial Cryptography*, volume 1648 of *LNCS*, pages 184–195, 1999.
31. S. Stubblebine, P. Syverson, and D. Goldschlag. Unlinkable serial transactions. *ACM Transactions on Information and System Security*, 2(4):354–389, November 1999.

Appendix A

Proof of Lemma 1:

Assume, by contradiction, that we use both positive and negative ranges to express a solution. We can show that a solution that uses k positive and negative ranges can be expressed using no more than k ranges of the positive type only.

Suppose we have a positive range r_1, which covers documents starting from r_{1s} up to r_{1f}, and a negative range r_2 from document r_{2s} to document r_{2f}, respectively. Then with respect of their relative position, there are four different cases when r_1 and r_2 overlap and we consider them one at a time[6]:

1. $r_{1s} < r_{2s}$ and $r_{1f} > r_{2f}$. In this case the intervals r_1 and r_2 can be successfully replaced with two positive intervals r_1' and r_2' that range over documents $(r_{1s}, r_{2s} - 1)$ and $(r_{2f} + 1, r_{1f})$, respectively.
2. $r_{1s} < r_{2s}$ and $r_{1f} < r_{2f}$. This case can be handled by a single positive interval with bounds $(r_{1s}, r_{2s} - 1)$.

[6] For the sake of simplicity, we assume that all of r_{1s}, r_{1f}, r_{2s} and r_{2f} are distinct. In cases when this condition cannot be assumed to hold, only structurally insignificant changes to the proof are needed.

3. $r_{1s} > r_{2s}$ and $r_{1f} > r_{2f}$. In this case, we can also specify only one positive interval that will cover the same documents as the original two. The interval we obtain here is $(r_{2f} + 1, r_{1f})$.
4. $r_{1s} > r_{2s}$ and $r_{1f} < r_{2f}$. Here no ranges need to be specified.

Now assume that a negative range overlaps with two or more positive ranges. We show that we do not benefit from having negative ranges in the case when a negative range overlaps two positive ranges. A proof for the general case when a negative range overlaps with more than two positive ranges can be achieved by repeatedly applying the argument that uses only two positive ranges and such cases are never optimal.

Assume that the two positive ranges are r_1 and r_2 with bounds (r_{1s}, r_{1f}) and (r_{2s}, r_{2f}), respectively, and the negative range is r_3 and covers documents r_{3s} through r_{3f}. Without loss of generality, assume that the positive ranges are non-overlapping (any two overlapping ranges can be replaced by one non-overlapping) and $r_{1f} < r_{2s}$. Then there are four cases of different relative positions of r_1, r_2, and r_3:

1. $r_{1s} < r_{3s}$, $r_{1f} > r_{3s}$, $r_{2s} < r_{3f}$, and $r_{2f} > r_{3f}$, which can be replaced by two positive intervals with ranges $(r_{1s}, r_{3s} - 1)$ and $(r_{3f} + 1, r_{2f})$.
2. $r_{1s} > r_{3s}$ and $r_{2f} < r_{3f}$, where all intervals can be simply dropped without affecting the result.
3. $r_{1s} > r_{3s}$, $r_{2s} < r_{3f}$, and $r_{2f} > r_{3f}$, in which case a single positive range $(r_{3f} + 1, r_{2f})$ can be used.
4. $r_{1s} < r_{3s}$, $r_{1f} > r_{3s}$, and $r_{2f} < r_{3f}$, in which case also a single positive range $(r_{1s}, r_{3s} - 1)$ can be used.

The case when two negative ranges overlap with a single positive can be proved using a similar argument as above and is omitted due to insignificant changes. Thus, it follows that any solution that uses k negative and positives ranges can be replaced by a solution that uses at most k positive ranges. This completes the proof. $\square$