PORTABLE AND FLEXIBLE DOCUMENT ACCESS CONTROL MECHANISMS

by Mikhail Atallah and Marina Bykova

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Portable and Flexible Document Access Control Mechanisms[*]

Mikhail Atallah, Marina Bykova
Computer Sciences Department and CERIAS
Purdue University

{mja,mbykova}@cs.purdue.edu

Abstract

We present and analyze portable access control mechanisms for large data repositories, in that the customized access policies are stored on a portable device (e.g., a smart card). While there are significant privacy-preservation advantages to the use of smart cards anonymously created and bought in public places (stores, libraries, etc), a major difficulty is that, for huge data repositories and limited-capacity portable storage devices, it is not possible to represent any possible access configuration on the card. If we let n denote the number of documents on a server, then we need to design succinct descriptions of portable access rights to arbitrary subsets of these n documents, such as they "fit" in only k available space, where k is much smaller than n. We describe and analyze schemes for both unstructured and structured collections of documents. For these schemes, we give fast algorithms for efficiently using the limited space available on the card. For a customer whose card is supposed to contain a subset S of documents, access to all of S must be allowed. In some situations a small enough number of "false positives" (which are accesses to non-S documents) is acceptable to the server, and the challenge then is to minimize the number of false positives implicit to any given card. In our model the customer does not know which documents correspond to those false positives, the probability of a randomly chosen document being a false positive is small, and too many unsuccessful access attempts are viewed by the server as an exhaustive search attack, which can possibly result in zero-ing out the card.

Recent related work by Bykova and Atallah was geared towards the situation where the document repository and/or access policies change rapidly, and are therefore not vulnerable to on-line sharing of false-positive experiences by different users. In this paper we seek to prevent such collusive attacks by different card holders: It is a design requirement that the information in one card is useless to the holder of another card; that is, even if two customers have the same S, they would not have the same set of false positives.

1 Introduction

1.1 Problem description

The problem of access control has been studied extensively over the past years. Several access control models have been developed and received a varying degree of attention and usage. In particular, role-based access control (RBAC) is in wide use due to its natural applicability to many environments, but it is not particularly useful in some settings, when, for instance, every user requires unique access rights. In this work we consider a very large collection of documents or

[*] 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.

other objects, and do not limit the customers to rigid menus of pre-defined sets of items to which they may have access. With the rapid recent increase in the number and the level of maturity of on-line document collections (digital libraries and the like) that provide payment-based access to their documents, this model has emerged as an appropriate way of dealing with this explosive growth. In order to make this model as flexible and convenient for the customer as possible, we want to allow each customer to choose a custom set of documents to be included in his subscription order.

As stated above, this model might not appear interesting or challenging to implement: each customer receives a unique policy configuration that allows him to access the documents requested. What makes this problem more intriguing are the additional requirements that we impose. More precisely, we wish to fit such customer policy configurations into a limited amount of space, yet still be able to provide access to large data sets. This requirement becomes important in our era of portable devices such as smart cards and sensors that are space and/or battery-limited.

One might argue that putting the access policies on limited-storage smart cards is not a sound approach to portable access rights, that one can instead store the access policies on the server end. For privacy reasons, instead of storing customers' real identity, the server and the smart card could then store only a random ID (a pseudonym) while the access policies associated with that ID are kept at the server. In this case, the customer gets privacy without any need for storing the access policies on the card itself. This, however, has the disadvantage that tracking and profiling of a random ID's document access patterns is still possible; the danger then is that, if even once the random card ID is associated with an event (e.g., access from a particular IP address) that is linked to the customer's real identity, the whole access history of that individual becomes known to the server. This problem does not occur when the access policies are on the card itself. As both a metaphor and an example, we envision a customer, who walks into a bookstore; selects a number of items from a large universe of books, articles and newspapers; pays with cash; and receives a smart-card that provides him private access to the items selected at many terminals in various locations at stores, libraries, etc.

Storing access rights on the card itself has another advantage in that the entity that issues the card can be distinct from the entity that provides access to the data. In other words, the customer policy configuration can be constructed and written on the card by the data owner, while access to the data is performed by a third-party data publisher (possibly not completely trusted). Digital Rights Management (DRM) opens another avenue for utilizing digital portable access rights that the card can carry (see [1] for an overview of DRM techniques). For example, a card can manage access to a large collection of multi-media objects while using only a limited local storage space.

As we attempt to reduce the storage requirements of access policies, we lose the ability to represent all possible subsets of the documents from the data collection in a deterministic way. Some documents or subsets of documents might have to share the same access policies which means that, given a configuration policy, a customer might unintentionally receive access to more documents that was originally requested; we refer to these accessible yet not requested (hence not paid for) documents as "false positives". In a competitive market, companies can clearly not charge their customers for such false positives, yet they will need to minimize them and the losses they cause. One of our main goals is therefore minimizing these additional "false positive costs" caused by the limited portable storage space. In order to prevent dishonest customers from sharing information learned about the false positives implicit to their card, we also require policy representations to be unique to each card, even if they correspond to the same set of documents. It is the constraints such as this minimization and the requirement of unique policy representations that make the problem very different from a mere data compression problem. Another (more minor) reason why we cannot use compression is that the access policies representations must be usable "as is" without uncompressing them first: There is no room in the smart card for that, and using server memory to uncompress would make profiling of the card's usage patterns possible, throwing us back into the lack of privacy-preservation that we sought to avoid.

1.2 Summary of results

Our contributions are as follows:

- For an unstructured collection of documents, we give and analyze a scheme that is based on generating a "good" random permutation for the documents in a subscription and is suitable for any arbitrary subset of the documents. We provide analysis of time complexity and cost of the scheme.

- For a structured collection of documents, we give a scheme similar to the random permutations one which has been modified to appropriately suit the structured data. We provide analysis of time complexity of the scheme.

1.3 Paper outline

The layout of this paper is as follows: Section 2 provides an overview of the prior work. Section 3 gives a detailed problem specification and requirements that are imposed on any resulting solution. In section 4, we describe the solution for an unstructured collection of documents and provide efficient algorithms for card generation. It also discusses the viability of the scheme, provides a sample policy, and analyzes the approach with respect to our design goals. In section 5, we describe a solution for the (more difficult) case of structured data, namely hierarchies, and provide algorithms for them as well. Finally, section 6 concludes the paper and gives directions for future work.

2 Related Work

Most literature on digital libraries does not explore the problem of access control, and many deployed systems nowadays provide only a single or otherwise a few inflexible subscription types. Payette and Lagoze [18] recognized this problem and proposed a way of solving it by introducing a spectrum of policy enforcement models which range from system-wide to object-specific and are capable of providing very flexible access control. They, however, give only a general framework for specifying policy enforcement mechanisms but do not deal with the problem of policies assignment itself.

Work conducted on XML [23] explores the problem of access control for on-line documents. Recently, there has been extensive work on securing access to XML documents and using XML as a tool for specifying security policies [5, 6, 7, 12, 13] (also, for a list of recent papers related to security of XML see [20]). Bertino et al. [4] use binary strings to represent both customer policy configurations and document policies in the third-party publisher model, i.e., the model in which the roles of the *data owner* (who assigns policy configurations to its customers) and *data publishers* (who provide the service and enforce the access control according to document policies) are separate. We consider binary strings to be the most space efficient and precise way of policy representation and adopt policies in the form of binary strings in our work. Bertino et al. [4], however, allocate one bit per policy on the assumption that there will be a limited number of different subscription types, and their approach becomes inefficient as the data repository grows in size and a customer is able to choose a customized and possibly unique document subscription set. Therefore, we need a method of representing policy strings so as to be able to store and use them on space-limited media.

The idea of achieving space efficiency at the cost of a small probability of false positives was introduced in Bloom [8]. The Bloom filter is a clever randomized data structure for concisely representing a set to support approximate membership queries and is widely used in a broad spectrum of applications ([9, 14, 17], to name a few). The queries never result in false negatives (i.e., a query result never says that an item is not a member of the set if it, in fact, is a member) but might result in false positives with a small probability, which is acceptable for many applications. This approach achieves a better space utilization than a simpler representation with one hash function, but even in the case of Bloom filters, the filter length (which in our case corresponds to the card capacity) should be larger than the total number of items in the set n to result in a reasonable performance. This is not suitable for the problem we are trying to solve and therefore the Bloom filter approach cannot be used "as is" for our problem. Customized Bloom filters also do not appear to provide acceptable results.

The problem of policy assignment of minimal cost for a set of homogeneous objects within a large data repository was explored in more detail in Bykova and Atallah [10], and their problem is very close to the one we are considering. The present paper adds two new dimensions to the problem, and ends up with very different solutions from [10]: (i) it considers hierarchically structured collections of documents, and (ii) it is inherently resilient to collusive attacks by users. In the schemes of [10], the policy assignment algorithm is static and the pattern of false positives is preserved over all subscription orders and customers. This means that once a customer has learned that a particular combination of documents results in free access to another document (false positive), he can enable a different customer with the same subscription set to also obtain the same false positive.

Recent techniques for the problem of software license management through portable limited-storage card-based access rights [3, 2] 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 [3, 2]).

Work on unlinkability and untraceability was started by Chaum [11] and has been explored more extensively in the recent years. In particular, work on unlinkability includes anonymous group authentication ([21, 16, 19, 15] and others) and unlinkable serial transactions [22] for subscription-based services. All of the prior work, however, does not account for the fact that descriptions of access rights (or service types) might be long and required to be portable. In this work, we describe a scheme that combines compact policy representation with transaction unlinkability.

3 Problem Specification

In this section, we list the problem specifications, and then the properties desired in any solution.

3.1 Specifications

The goal is to design an efficient access control scheme for the following specifications:

1. We are given a large data collection of n items (documents, multimedia objects, etc.).

2. A customer can request access to any subset of the items in the repository (m documents) and is charged according to the set of documents selected.

3. Access policy configuration is stored on a card of limited capacity of k cells, each cell having $O(\log n)$ bits such that $k \log n < n$ and $k < m$[1].

[1] In cases when $k \geq m$, the list of selected documents can be explicitly stored on the card.

3.2 Goals

Below are the goals that guide our design decisions:

1. **Low rate of false positives**. One of our main goals is to design a scheme with a reasonably low rate of false positives meaning that, given a subscription order for m documents, the number of documents m' that the customer can unintentionally access for free is bounded by some suitably low value. The threshold could be defined as a function of m', m, and/or n, but in all cases must be tolerably low to the service provider.

2. **Transaction untraceability**. For customer privacy, transactions should not be linked back to the customer who is making a request to access a document.

3. **Transaction unlinkability**. To provide further customer privacy, two transactions by a single customer should not be linked together, thus making customer profiling impossible. This desired property does not allow us to permanently store complete customer orders at the service provider, and it is desirable that an order is processed by a separate entity (e.g., the data owner) and discarded after it has been processed.

4. **Unforgeability**. It should be impossible for an entity other than the legal service provider to issue cards that will allow access to the document repository. This means that all terminals that read smart cards will ask a smart card to present evidence of authenticity of the card (evidence that only the service provider can initially issue).

5. **Unique policy representation**. In order to lower damage caused by dishonest customers that collaborate to discover as many free documents (false positives) as possible, we would like our scheme to be resistant to such behavior. This requires not only that false positives depend on the subscription order but they uniquely differ from one request to another even when the set of requested documents stays unchanged over multiple orders. With such a scheme in place, no correlation between free documents in different orders is possible, and any gain to customers who collude is eliminated. All a dishonest customer can do is to try to discover for himself the free documents for his particular order, which, combined with some penalty mechanism, can be prohibitively inefficient for him to do.

6. **No additional sources of information**. All information needed to perform access control verification should be stored on the card itself. No other sources of additional storage may be required (such as storage space of a home workstation in case of software license management), as there is no single place where such information can be stored.

7. **Fast access verification**. Policy enforcement and access verification should be performed in real time and therefore cannot experience delays due to expensive computations.

8. **In-house card generation**. The card generation process should be relatively short. It might be performed on a powerful computer, but within a certain time limit. For example, a card can be generated while the customer is waiting in a line for the cashier at a bookstore.

   A case can be made for relaxing this last constraint if there exists a scheme that requires an extended amount of time for card generation but is much more efficient and less costly (compared to other models that obey this constraint). In this case, a card is delivered to the customer when its processing completes (like an order at a pharmacy, that usually involves a wait).

9. **Forward compatibility.** An old card does not lose its validity as the repository grows.

4 Solution for Unstructured Data

We now present our approach for an unstructured collection of documents and provide its analysis. In what follows and in the rest of this paper we use to 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.

Our solution consists of generating random permutations of documents included into 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 the subsequent permutation of the documents, we run the 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; and a new permutation is generated and tested otherwise. Information written on the card includes the 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 (i.e., not from the original order) as possible. Consider an oversimplified example where the repository has a size of 10, our card can store 2 intervals, and we receive a customer subscription order for documents 1, 5, 7, and 9. Suppose that after permuting the documents we obtain set {2, 3, 6, 8}, so the best option in this case is to use intervals 2–3 and 6–8 for storing the set on the card. The cost of a solution is computed as the number of false positives (in the example above, the cost of the permutation is equal to 1).

Both the random permutation seed and the document intervals are a subject to the card's storage constraints. Since a smart card's capacity is $O(k \log n)$, we can use it to store $O(k)$ numbers within the range $\{1, \ldots, n\}$, or k intervals. The permutation seed can be up to $O(k \log n)$ 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, negative ranges do not improve the result by decreasing the cost of a solution, as the lemma below shows (but, as we show later on, 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. □

In the rest of the this section, we use $r = \{r_1, \ldots, r_m\}$ to compactly represent a customer order of m documents. Each r_i uniquely identifies a single document in the repository (i.e., it is a number in the range $\{1, \ldots, n\}$) and all r_i's are sorted in increasing order such that $r_i < r_{i+1}$ for $1 \leq i < m$. We first present an algorithm for producing a suitable encoding to be placed on a card (given in 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 (discussed in 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 the next section (section 5) provides an extension to it that covers structured data.

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 here takes a subscription order of m documents and a set of rules, which 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 smart card, which consists of a permutation seed s and k intervals that optimally represent the requested documents r. Below is a more formal description of the algorithm.

Input: The repository size n, a customer order of m documents $r = \{r_1, \ldots, r_m\}$, and a set of stopping criteria $\tau = \{\tau_1, \ldots, \tau_t\}$.

Output: A seed s for generating a permutation and k intervals to be stored on a smart card.

Algorithm 1:

1. Seed the permutation algorithm with a random number s.
2. Permute the m documents to get $p_i = \pi_s(r_i)$ for each document $r_i \in r$.
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, and if a sufficient subset of them $\tau' \subseteq \tau$, $1 \leq |\tau'| \leq t$, is satisfied, output the solution. Otherwise, go to step (1).

The asymptotic bound of a single run of the 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 $O(n^k)$ loop invocations in the worst case, but typical values are lower. This time is constrained by the space available for storing a random seed s: there are $O(2^{k \cdot \log n}) = O(n^k)$ 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 k 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 $k - 1$ 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: k 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}. For example, c_1 is computed as $c_1 = p_2 - p_1 - 1$.

2. In $O(m)$ time select a $(k-1)^{\text{th}}$ largest among $c_1, \ldots, c_{m-1}$ (suppose that it is c_j).

3. In $O(m)$ time go through $c_1, \ldots, c_{m-1}$ and choose $k-2$ entries that are $\geq c_j$. Those entries and c_j correspond to the $k-1$ "gaps" between the optimal k intervals, i.e., they define the optimal k intervals.

Note that the "cost" of the solution is $C = c_1 + ... + c_{m-1} -$ (sum of the largest $k-1$ 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 such method that has the properties listed below should be suitable to use with 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 $O(n^k)$ of these permutations[2]. This turns out to be enough in practical situations (cf. 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, $1 \leq x \leq n'-1$: Any integer i, $1 \leq i \leq n'-1$, would then map into $\pi_x(i) = x * g^i \bmod n'$. 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 seed means that only n of the possible permutations of n elements are "accessible". To extend the "reach" of the seed from only n permutations to the full n^k allowed by the available $O(k \log n)$ bits of storage, we would simply store as a seed k distinct (rather than a single) such x values: $x_1, \ldots, x_k$. Each x_j defines a permutation π_{x_j} in the manner described above: For the jth such permutation, i maps into $\pi_{x_j}(i) = x_j * g^i \bmod n'$. The "true" permutation described by this seed of length $k \log n$ bits, is then the functional composition of the permutations $\pi_{x_1}, \pi_{x_2}, \ldots, \pi_{x_k}$ (in that order). There are n^k 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^k], could be used for our purpose of permuting: If x is the seed, then $\pi_x(i)$ is simply the encryption of i using x as key. That n is too small for cryptographic security is not an issue here, because we are not using encryption to hide, only to permute.

[2] 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 described above refer to the card generation process, 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. We only briefly note that possibilities include sharing secret keys with the server (a secret key is not unique to a card) and/or using other means of low-computation anonymous authentication that is suitable for smart cards.

Policy enforcement by using the policy encoding placed on a card is performed 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 k 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 k intervals for p_i to determine whether the value is covered by one of them or not. Since we can sort the k intervals before storing them on the card, this operation can be performed in $O(\log k)$ time using binary search.
- If p_i is covered by one of the k intervals, the card requests the document i from the server. Otherwise, it notifies the user about access denial.

One can see from the description above that untraceability and unlinkability constraints of our design (goal (2) and (3) in section 3.2) are satisfied: Each card anonymously authenticates itself to the server and does not send any information to 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 (goal (6)) and uses an efficient method for such enforcement (goal (7)).

4.5 Economic Analysis

This section analyzes the practicability 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" mentioned in the previous section that govern permutation selection process more precise.

4.5.1 Values of interest

As input, we are given the size of data repository n and the number of documents in a customer order m[3]. Other parameters of use for determining what an acceptable cost is are:

$c_{card}(m)$ – price the customer paid. This value represents the price charged for an order of m documents and is a (possibly arbitrary) function of the documents that comprise the order.

$t(m)$ – maximum number of unauthorized document requests allowed. Each card can keep information about 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

[3] In reality, we have the entire order $r = \{r_1, \ldots, r_m\}$ 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 item to another. In this case, each derived value that takes m as a parameter can be computed as a function of the set r itself.

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. In order for this mechanism to work, each customer should be informed about t at the time of purchase of the card and should be given an explicit list of the documents included into his order.

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

$n'(n, m)$ – 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 scenario, any free document can be viewed as valuable to the attacker. In the best case, the attacker has zero interest in anything outside the m documents he 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 and determine when to stop generating new permutations.

1. **Threshold for the number of false positives** m' 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:

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

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 the 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. **Timeout**. Under some policies, the card generation process might have to be carried within a certain period of time. In this case, if no suitable permutation is found during the interval, the best permutation tried so far is used to create the card.

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 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 (2)–(7) listed in section 3.2 are met. Goal (9) 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 (goal (1)), 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 (8). If the service provider employs a policy that includes a timeout, then obviously in-house card generation is always achievable. If the service provider, however, places more weight on minimizing the number of false positives, then this constraint might be relaxed.

5 Structured Data

This section explores the possibility of extending our approach to structured data such as trees. In many data repositories documents are stored in hierarchies, which makes it possible to utilize the repository structure and reduce the number of false positives in the solution computed.

5.1 Tree structure

Suppose we are given a tree of n documents and a subscription order of m documents. The card's capacity is still $O(k \log n)$ bits or $O(k)$ 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 (= lower in the tree) wins. Finally, the word "cost" in the rest of this section is used as "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.

As with any dynamic programming approach, the cost of an optimal solution at any given node v needs to be calculated for a number of 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 $0 \leq j \leq k$.

Here we provide an algorithm for binary trees, which can naturally be extended to work for more general t-ary trees with $t \geq 2$. 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.

5.1.1 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 the value of $C^+(v,\ j)$ is computed as:

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

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 $C^-(v,\ j)$: Similarly, for the negative context there are 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:

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

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:

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

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 $C^-(v,\ j)$, in this case we use:

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

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

5.1.2 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; 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.

5.1.3 Complexity analysis

To compute the cost of an order, we use the above rules to compute $C^{-}(root, k)$. 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 k slots, this algorithm runs in $O(n \cdot k^2)$ time for binary trees. For arbitrary t-ary trees the algorithm gives $O(n \cdot k^t)$ time.

5.1.4 An extension to records of variable depth

Let h be the height of the tree. The dynamic programming approach we have 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 in the paper due to space considerations.

For a t-ary tree, this modification implies a factor of h (but not h^t) because any record placed at the parent covers one child's subtree at same depth as for another child's subtree. Thus, this adds an extra h to the time complexity.

5.1.5 Note

Currently, the algorithm presented above is static because no permutation for the tree structure is used. In order to make this scheme viable, more research needs to be performed to introduce uniqueness of each solution through means other than permutation, e.g., false positives are randomized for each order but are kept below a certain threshold.

6 Conclusions and Future Work

In this work we presented a problem of fine-grained document access control under space restrictions. Our solution preserves customer anonymity, uses efficient algorithms to perform access control, and at the same time minimizes loss caused by policy compression. We gave a full-grown solution for unstructured data and provided a method for evaluating the cost of a solution for hierarchically structured repositories. Future directions include providing more thorough (possibly empirical) analysis of our scheme and building a solid framework for hierarchical data.

This work can be extended to cover other types of structured data. In particular, grids can be of practical interest. Grid structures can be useful in the context of Geographic Information Systems (GIS) subscriptions where land is partitioned into cells of a standard size. A customer can subscribe to a cell and receive information about temperature, humidity, precipitation, and other meteorological data relevant to the area. Each subscriber selects cells of his interest and pays to get access to a customized area of his choice. Access control is enforced through the use of cheap cards of limited capacity. An algorithm to compute the optimal cost of a subscription in this case will model geometric algorithms for approximate representation of a polygon. A difference from the standard approximation methods here is that the requested area must be included entirely in the card, while the number of other cells stored on the card should be minimized.

Acknowledgments

We would like to thank anonymous reviewers of ESORICS 2004 for their valuable comments.

References

[1] M. Atallah, K. Frikken, C. Black, S. Overstreet, and P. Bhatia. "Digital Rights Management," *Practical Handbook of Internet Computing*, Munindar Singh (Ed.), CRC Press, 2004.

[2] M. Atallah and J. Li. "Enhanced Smart-card based License Management," *IEEE International Conference on E-Commerce (CEC),* Jun. 2003, pp. 111–119.

[3] T. Aura and D. Gollmann. "Software license management with smart cards," *USENIX Workshop on Smart Card Technology*, USENIX Association, May 1999.

[4] 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.`

[5] E. Bertino, S. Castano, and E. Ferrari. "On Specifying Security Policies for Web Documents with an XML-based Language," *ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, May 2001.

[6] E. Bertino, S. Castano, and E. Ferrari. "Securing XML Documents with Author-$\mathcal{X}$," *IEEE Internet Computing,* Vol. 5, No. 3, pp. 21–31, 2001.

[7] E. Bertino and E. Ferrari. "Secure and Selective Dissemination of XML Documents," *ACM Transactions on Information and System Security,* Vol. 5, No. 3, Aug. 2002, pp. 290–331.

[8] B. Bloom. "Space/time trade-offs in hash coding with allowable errors." *Communications of the ACM,* Vol. 13, No. 7, pp. 422–426, 1970.

[9] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey," *Allerton Conference,* 2002.

[10] M. Bykova and M. Atallah. "Succinct Specifications of Portable Document Access Policies," *ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, Jun. 2004.

[11] D. Chaum. "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM*, Vol. 24, No. 2, Feb. 1981, pp. 84–88.

[12] 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,* Vol. 5, No. 2, May 2002, pp. 169–202.

[13] P. Devanbu, M. Gertz, A. Kwong, C. Martel, and G. Nuckolls. "Flexible Authentication of XML Documents," *ACM Conference on Computer and Communications Security (CCS'01)*, Nov. 2001.

[14] L. Fan, P. Cao, J. Almeida, and A. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking,* Vol. 8, No. 3, pp. 281–293, 2000.

[15] J. Kim, S. Choi, K. Kim, and C. Boyd. "Anonymous Authentication Protocol for Dynamic Groups with Power-Limited Devices," *Symposium on Cryptography and Information Security (SCIS'03)*, Vol. 1/2, pp. 405–410, Jan. 2003.

[16] C. Lee, X. Deng, and H. Zhu. "Design and Security Analysis of Anonymous Group Identification Protocols," *Public Key Cryptography (PKC'02)*, LNCS, Vol. 2274, pp. 188–198, Feb. 2002.

[17] M. Mitzenmacher. "Compressed Bloom Filters," *ACM symposium on Principles of Distributed Computing*, Aug. 2001.

[18] S. Payette and C. Lagoze. "Policy-Carrying, Policy-Enforcing Digital Objects," *Research and Advanced Technology for Digital Libraries, 4th European Conference (ECDL'00),* Vol. 1923, pp. 144–157, 2000.

[19] P. Persiano and I. Visconti. "A Secure and Private System for Subscription-Based Remote Services," *ACM Transactions on Information and System Security*, Vol. 6, No. 4, Nov. 2003, pp. 472–500.

[20] C. Geuer Pollmann. The XML Security Page, `http://www.nue.et-inf.uni-siegen.de/~geuer-pollmann/xml_security.html`.

[21] S. Schechter, T. Parnell, and A. Hartemink. "Anonymous Authentication of Membership in Dynamic Groups," *Financial Cryptography*, LNCS, Vol. 1648, pp. 184–195, 1999.

[22] S. Stabblevine, P. Syverson, and D. Goldschlag. "Unlinkable Serial Transactions," *ACM Transactions on Information and System Security*, Vol. 2, No. 4, Nov. 1999, pp. 354–389.

[23] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (second edition), October 2000, W3C Recommendation, `http://www.w3.org/TR/REC-xml`.

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[4]:

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 $(r_{1s}, r_{2s} - 1)$.

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.

[4] 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.

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. □